The Apple Macintosh as a User Interface Agent for Unix Systems

---

A thesis
submitted in partial fulfillment
of the requirements for the Degree
of
Master of Science in Computer Science
in the
University of Canterbury
by
J.R.A.Collier

University of Canterbury
1988

Table of Contents

Acknowledgements

I owe grateful thanks to my supervisor, Dr. Mike Maclean, for his advice and comments on the thesis, and also for his patience during the prolonged "final stages" of the project. I should also like to thank his predecessors, Dr. Bruce Mackenzie, Dr. Dick Cooper and particularly Robert Biddle, who first proposed the project and gave valuable advice and support throughout its duration, although he was unable to continue in the rôle of supervisor. Greg Ewing also deserves special acknowledgement for his comments and suggestions which helped shape MPFW.

Abstract

This thesis investigates the possibility of using the Apple Macintosh to provide a workstation-like interface for UNIX hosts. Problems of UNIX interface development are identified through a study of existing UNIX interface systems. Techniques from some of these are applied in the design and implementation of MPFW - the Macintosh Programming Framework for Workstation Interfaces. This system provides communications and graphical interface support for developing application-specific interfaces on the Apple Macintosh.

Part of the thesis is devoted to evaluating the programming facilities offered by the Macintosh. Several weaknesses are identified, and mechanisms for redressing these are featured in MPFW to simply interface programming. Pre-emptive multitasking is one such mechanism, and its implementation is described in some detail.

Several related support projects are also presented, including a pseudo-terminal server daemon for Berkeley UNIX systems.

Chapter 1

The Macintosh as an Interface for UNIX Systems

1.1 Introduction

This thesis is primarily an evaluation of the Apple Macintosh™ and its suitability as a user interface agent for a UNIX™ host. Emphasis is given to programming issues which arise in such a distributed system, particularly the difficulties of writing small Macintosh interface programs and using them concurrently to allow multiple application-specific interfaces. A prototype support system which addresses these issues is presented.

The project was motivated in part by the machines in use at the Department of Computer Science: the Department runs 4.3BSD UNIX on a VAX™-11/750 as its central host for research and undergraduate teaching, and Macintoshes are used by staff and students as terminal emulators and general purpose microcomputers. Recently the Department acquired a number of Sun/3 UNIX workstations and an Ethernet local area network, and the Macintoshes are now connected by a PhoneNET system with a gateway to the Ethernet through a Kinetics FastPath™ bridge.

At the inception of the project, however, these Macintoshes communicated with the host via RS422 serial lines, and the systems discussed in this thesis employ this mode of communication. It was considered then that the simple single-windowing terminal emulators which were available for Macintosh to UNIX communications made poor use of the Macintosh's capabilities, and that it would be worthwhile to try to present a graphical interface to UNIX using the Macintosh interfacing standards.

Since then, this topic has received much attention from commercial developers and in universities, and a number of interface systems have been produced [uw, MacNix]. These systems have yet to integrate the Macintosh graphical interface at

---

™ 'Macintosh' and 'Apple' are trademarks of Apple Computer Inc.

™ UNIX is a trademark of AT&T Bell Laboratories.

™ VAX is a registered trademark of the Digital Equipment Corporation

™ Kinetics and FastPath are registered trademarks of Kinetics, Inc.

the level of specific applications, but their generalised interfaces are nevertheless as ambitious as was originally intended for this project. It is unlikely that a single style of interface will be suitable for all applications and classes of user. The focus has therefore shifted from development of a pre-programmed workstation to the implementation of a framework system which provides high-level communications and Macintosh development support for producing customised distributed interfaces.

Part of this thesis is devoted to a general investigation of methods for integrating UNIX and the Macintosh, and the facilities which should be provided for the programmer in such a distributed environment. The two systems exemplify opposite priorities in operating system design: UNIX gives precedence to architectural regularity at the expense of the user interface; the Macintosh, *vice versa*. There are underlying architectural considerations which must be considered carefully when designing a user interface for UNIX, and there are also limitations in the Macintosh programming environment which need to be addressed in order to assist experimental interface programming. The programming projects to be described in this thesis have been designed to explore ways of combining the advantages of both systems, either by using them together in an integrated system, or by importing some of the features of one into the programming environment of the other.

1.2 Sub-projects Contributing to the Research

1.2.1 Macintosh Programming Framework for Window interfaces

The largest part of the project has been a program which acts as an extra layer of management software between the Macintosh operating system and a running application, reducing the need for programmers to support operating system and low-level window management functions. An important feature of MPFW[1] is the implementation of multiple scheduled processes above the Macintosh operating system. Process management above an operating system which is not re-entrant has entailed considerable difficulties, but this design has been found to be effective in isolating the heterogeneous components of a window interface system.

---

[1]originally "Monty Python's Flying Workstation"

Most existing user interface management systems for UNIX (§2.3) require that any application which is to use a structured window interface (i.e. not a simple terminal emulator) be re-written on the host. The intention of MPFW is to provide a vehicle for experimenting with interfaces tailored for existing host applications. These interfaces should be designed to run on the Macintosh without modifying the host applications, directing output to the appropriate portion of the application's window, and generating textual input for the application in response to the user's keystrokes and interaction with Macintosh controls and dialog boxes.

MPFW was designed to ease the programming of these interfaces by automating features which most would hold in common, including window management and the low level host-to-Macintosh communications. These enhancements to the Macintosh operating system solve some of the problems involved in Macintosh programming and may prove useful beyond the scope of UNIX interface design. Chapter 5 discusses the structure of MPFW in detail.

1.2.2 Host ↔ Macintosh Serial-line Protocol

Unlike a simple terminal emulator, a graphical user interface system such as MPFW may pass large quantities of data without immediate echoing. A multiplexing capability is also required to support multiple windows displaying output from separate host processes. For these reasons, it was considered necessary to run a reliable communications protocol over the serial connection between the host and the Macintosh. Such a protocol has been integrated into MPFW. Its design and implementation are described in Chapter 6.

1.2.3 Interface Programming Language

It was intended that interface programs for MPFW should be written in a small, purpose-specific language. This would have allowed MPFW to remain independent of proprietary programming packages, as well as promoting a program structure which was better suited to the original, strictly event-driven model of interface sub-applications. The latter justification became less relevant when the viability of pre-emptive multitasking was confirmed in MPFW, and in view of the difficulty of producing a system which could compete with commercial compilers for well-known languages, the project was abandoned. Appendix IV outlines the prototype of the language and its compiler, and the means by which programs were integrated into earlier versions of MPFW.

1.2.4 A Pseudo-Terminal Server Daemon for BSD UNIX (PTYD)

BSD UNIX provides devices to support multiple-window interface systems, but the means of obtaining ownership of these devices has adverse security ramifications and also adds some complexity to window management programs. PTYD, as described in Chapter 7, addresses both these problems. A copy of the program has been sent to the University of California at Berkeley to assist in the development of a generalised resource server for the next release of BSD UNIX.

1.2.5 Programming Support for Macintosh and UNIX projects

A number of support tools have been developed to assist in the management of the major thesis projects above. Chapter 4 gives a brief description of these tools. Many of the projects required compilation sequences with components on either machine; the interface programming language, for example, used a Macintosh-resident development system in conjunction with a parser generator on the host. In other cases, general purpose UNIX programming tools were used to aid the development of C programs on the Macintosh.

At the core of this distributed development environment is the communications program Macscrewn - a customised single-window terminal emulator based originally on Dr. Bruce McKenzie's TVI925 emulator Macscreen. Support of the Macscreen was taken up early in the project as an exercise in developing skills in Macintosh serial communications programming. Since then it has gradually been rewritten, and its name has been altered in recognition of the idiosyncrasies exhibited by early experimental versions. Performance gains and added features encouraged appropriation of these versions before they were fully tested.

The relevance of Macscrewn to the project extends beyond the skills acquired during its development. It was important to have a stable, well-known application for testing the methods applied in MPFW's interface management and serial communications. This was accomplished by replacing modules of Macscrewn with their extended MPFW counterparts, or by slotting in new MPFW modules and invoking them through menu selections and escape sequences from the host. Some of the interface ideas which were designed for MPFW sub-applications were incorporated into Macscrewn in this manner.

Several extensions to Macscrewn were required for its rôle in communications support for the distributed development environment. These extensions include script-driven file/folder transfer facilities, clock synchronisation, file status enquiry support and remote application launching. Using Macscrewn and a number of small, non-interactive file manipulation applications, many laborious tasks - file backups, library management, compilation sequences, and compilation dependency calculations - have been automated through UNIX shell and Megamax BatchX command scripts. While this environment was implemented primarily to assist the Macintosh programming in the project, it nevertheless makes a useful study of an alternative approach to integrating the two systems.

Chapter 2

User Interfaces For UNIX

2.1 Background to UNIX

The UNIX operating system was originally developed by Dennis Ritchie and Ken Thompson at Bell Laboratories in 1973 [BELL 78]. Since then it has achieved great popularity with programmers, perhaps not least because the source code has been available to educational institutes and many have used UNIX as a model for the teaching and research of operating systems. Many computer manufacturers now support UNIX, often as an alternative to their own proprietary operating systems, and a number of variants have appeared over the years. Parts of this thesis are applicable only to those systems deriving from work at the University of California at Berkeley, known as the Berkeley Software Distribution (BSD) systems. Other major variants include the AT&T commercial releases, current of which is System V Release 3, and the Bell research systems known by the edition number of the manual (currently 8th and 9th).

The strength of UNIX lies less in the provision of highly developed monolithic applications than in the facilities which the operating system kernel and core utilities provide for writing and combining small programs to automate complex tasks. In order to use UNIX effectively, the novice is faced with some overheads in learning the functions of these utilities and the notational conventions they share, and also in acquiring a mental model of the operating system to guide the invocation of complex command sequences. For command line compactness and speed of typing, utility names are commonly two to four characters long; because of UNIX's development history, the mnemonic value of these names tends to be esoteric. Unsurprisingly, these pre-requisites often deter the general user. While computer professionals may find UNIX unsurpassed as a programming environment, others complain of "user-hostility".

Programming is an activity which requires intense concentration. If a development environment forces a programmer's attention towards time-consuming mechanical tasks, the thread of concentration may well be lost between periods of actual programming. Many such tasks are necessary during program development. Consider:

- control of the dependencies between multiple source files in order to determine those sections of a large program requiring re-compilation when sources are updated. This is especially important in large projects and in cases where some components are derived from automatic source generation tools.
- revision maintenance and source cross-indexing in large scale development projects
- consultation of reference material, which may be presented formally with dedicated retrieval systems (on-line documentation) or scattered throughout the file system (as in the case of sample system code or saved news articles).
- adaption and re-formatting of existing information - e.g. imported source code, system configuration and status tables, and output from standard utilities.
- maintenance of a customised user environment, including personal programming tools (libraries, command scripts etc) and preference selections for standard utilities. This must usually take into account such externally imposed factors as changing hardware configurations, different terminal types and work locations, and different machines in a distributed environment.
- preparation of program documentation and user manuals.

It would be impracticable to present monolithic applications which would automate these tasks to the satisfaction of most programmers, although there are high-level tools in UNIX which go far in providing automation of certain tasks (e.g. MAKE [FELDMAN 79] for dependency control). Instead, the UNIX programmer uses standard utilities - pattern searchers, stream editors, command interpreters and many others - to create customised tools as they are required. Most of these will be created as one-off interactive shell commands. A few will be seen to have general application and will be saved as command scripts, perhaps to be used later as components of still higher-level tools.

Consider an example where a section of code under development requires the use of a library call or a short algorithm, but the programmer does not remember the name of the library call or the source of the algorithm. In order to maintain productivity, the information must be available to the programmer within a time which will not break his or her attention to the code. In the case of a half-

remembered library call, the editing session may be interrupted with a couple of keystrokes and a short command will initiate a keyword search of the on-line manual, probably providing the library routine's entry within a few seconds. For the latter case, the programmer would launch several background process groups, each involving a tree-walking utility and a pattern matcher, to search for the required code in appropriate subsections of the file system. If similar searches are used frequently, the programmer may decide to encapsulate this sequence of commands in a script. Future searches can then be made simply by invoking the script and passing the code pattern as an argument.

The popularity of UNIX is not only a reflection of the versatility of its standard utilities. UNIX is designed to be portable; its kernel and core utilities are written in a high-level language, and the operating system is not based around any proprietary hardware system. The elegance of its kernel architecture makes the operating system easy to comprehend and to program both at application and system level.

While a detailed description of UNIX architecture is beyond the scope of this thesis, some grounding may be necessary in order to appreciate the factors which impinge upon the design of a command interface. Appendix A gives an introduction to some relevant features, and further references are provided at the end of the text.

As a programming environment, UNIX has probably been successful because it was designed by programmers for their own use, not for release as a commercial product. Despite the many additional features which have been added for general use in modern variants of UNIX, it has for the most part retained the simplicity and flexibility which programmers find desirable in adapting the operating system for their needs.

2.2 Problems in Designing a User Interface for UNIX

At an early stage, one of UNIX's main intended rôles was as a prototyping and cross-development tool. Unfortunately, perhaps, many of the features which aid programming for UNIX systems - the libraries, system call interface, and core utilities - are heavily dependent upon the underlying structure of the operating system. Cross-development is rarely viable, so the result has been that general users are foisted with an operating system which is in some ways inappropriate for their needs.

For although various windowing systems have been produced and some experimental menu-based shells for novices have been developed, efforts to provide an interface suitable for general users have met with only limited success. In part this failure may perhaps be attributed to developers' desire to preserve the regularity of the operating system, by attempting to fit the schemes into existing frameworks which are not entirely suitable.

One of the tenets of the UNIX "philosophy" is device independence, and this has therefore been one of the design goals of many user interface management systems - they have aimed at a general framework in which a variety of workstation designs can be accommodated. It is possible, however, that the design of an effective man-machine interface relies on psychological details which must be tailored minutely to a particular style of terminal in the context of any given application. This project looks only at the Macintosh, which has its own well defined style of interface, and no attempt is made to incorporate other workstations or window models.

While the flexibility of UNIX's standard I/O conventions is beneficial to the programmer, it discourages highly interactive application design. A compromise must be made between the needs of a human reading the output of a program and those of another program which may act on the same output. *Filters* are often included in pipelines when the format needs to be modified for a particular purpose. For example, a streaming text processor (e.g. SED [MACMAHON 79], AWK [AHO 79]) might be used to discard column labels from the output of a system status command before the information is passed to a monitoring program. *Pagers* are often appended to interactive pipelines so that the human reader can control the rate of output. In order to make these filters practicable, the information must not be heavily formatted, and complex input device interaction cannot be embedded in the core application.

The interfaces proposed in this project are tailored around individual existing UNIX applications. Each interface generates input for the application and directs output to a formatted window, relying on a knowledge of the application's behaviour to maintain the required contextual information. Simple, regular input and output formats are an advantage for this interface model as they are for filters. Whether it is possible to build an effective interface around a general-purpose

application is still an open question, but it is to be hoped that the systems developed in this project may help provide some answers.

2.3 Survey of Existing Interfaces

The first part of the project involved a study of various alternative UNIX interfaces, including a number of menu and window-based systems. The brief descriptions below include systems which were studied initially, and some (UW, NEWS) which became available during the development of the project.

2.3.1 Character-Oriented Terminal Interfaces

The first group consists of interfaces designed for terminals with little or no graphics capabilities. Their restrictions preclude fair comparison with bit-mapped display interfaces, but the models have been influential in interface design, and the structure of the windowing systems provided some guidance in the development of MPFW's window multiplexor.

2.3.1.1 MENUNIX

MENUNIX was developed in 1981 by Gary Perlman of the University of California at San Diego. This system is intended primarily for novice users, and presents the file system and utilities through a hierarchy of menus. The interface is designed for terminals without graphics capabilities. Menu choices are given by single keystrokes, indicating command, menu or file selection according to context. Ordinary textual commands may also be given. Standard utilities are grouped into menus (*workbenches*) by function (word-processing, programming etc.) rather than by location in the directory tree.

An implementation of MENUNIX was available for testing on the department's VAX. Although the supporting research described in [PERLMAN 81] was of interest and the system might have been of use in other environments, it seemed that there were several drawbacks to MENUNIX. Working through the workbench and file system menu hierarchies was time-consuming, especially as each menu change took around 15s even under light loading. Some of the workbench groupings also seemed counter-intuitive, and continual loss of screen contents as benches were changed removed the sense of continuity offered by conventional interfaces.

2.3.1.2 VSH

This was an early attempt by D.M. Sheibelhut to implement a menu-driven UNIX interface [SHEIBELHUT 79]. Like MENUNIX it is designed for text terminals. The current directory is displayed in a menu and files are selected by single keystrokes. A limited set of single keystroke commands is provided, but there are no "workbench" menus and no attempt is made to supply the full set of standard UNIX commands.

2.3.1.3 The Maryland Window Systems - WSH

This package, developed at the University of Maryland in 1983 [WEISEN 83], offers a library of C and LISP routines for providing menu and window interfaces for applications, as well as a shell - WSH - which allows multiple overlapping windows on text terminals.

Each WSH window is bordered by lines of a given character, and each provides an independent pseudo-terminal in which standard utilities can be run. Window creation and manipulation is performed by issuing commands which are linked to the shell.

WSH and a similar system WM (see below) seem sufficiently fast and well-developed to demonstrate the advantages of multi-windowing interfaces. For example, when using a debugger it is most useful to have source files visible at the same time as debugger output. In a shell or other interactive utility it is also helpful to be able to consult the on-line manual without breaking command input. More often, however, the time involved in setting up windows and manipulating them using textual commands is unwarranted, and the screen size available for each window in a character-oriented terminal is a serious limitation.

2.3.1.4 WM

This system, developed by R.K. Jacob in 1980 [JACOB 80], is very similar to WSH. Window manipulation commands are given by two character escape sequences. The original version was considered unacceptably slow, but an updated one [TRUSCOTT 85] has been installed on the VAX and used regularly by at least one member of the Department's technical staff.

2.3.2 UNIX Workstation Interfaces

A number of proprietary window management systems have been developed for UNIX workstations. Although some useful ideas were taken from the study of these systems, a close copy was considered inappropriate for the Macintosh project, since the underlying application is usually running on the same machine as the interface controller, or at least in a standard environment of communicating processes where machines are networked. It did not appear viable to attempt to port the UNIX subsystems which would have been required to support a similar structure for a Macintosh workstation.

Also, the interface models are based upon the capabilities the vendors' hardware, and the differences between these and the Macintosh (e.g. small screen vs. large screen, single-button vs. multi-button mouse) might make such an implementation clumsy. Besides this, much of the interface automation in the Macintosh ROM could not be applied, which would result in a greater programming burden and would alienate the resulting system from the body of conventional Macintosh software.

2.3.2.1 Suntools

The SUN windows system [SUN 86a] comprises several standard utilities ("tools") with their own graphical interfaces:

- a text terminal emulator, (SHELLTOOL) with a cut-and-paste facility
- a more sophisticated shell window (CMDTOOL) with a scrolling capability and improved command editing
- a graphics window tool
- a mouse based editor, similar to Macintosh editors
- a multi-window debugging tool
- various non-interactive tools such as a graphical performance monitor and clock

As well as these standard tools, SUN UNIX provides libraries of routines for implementing applications with graphical interfaces, including support for the underlying Suntools windowing system (SunView) and a range of graphical standards including GKS. The Suntools interface is in some ways closer to the Smalltalk [GOLDBERG 84] than the Macintosh, employing a three-button mouse and

hierarchical pop-up menus. Overall, the Suntools interface is more complex to manipulate, but judging by the tools provided it seems to have been designed to assist the more sophisticated UNIX user rather than as a novice interface.

2.3.2.2 AT&T 5620 - the BLIT terminal

The AT&T 5620 Dot Mapped Display [PIKE 84] is of interest as one of the earliest UNIX diskless workstation systems - it has a powerful processor and its own UNIX-like operating system but no provision for secondary storage, and it is always used in conjunction with a remote host. Graphical applications run as co-operating processes on the host and on the display, with the workstation code downloaded from the host. This design is in some ways similar to that of MPFW, except that in MPFW the interface code is stored locally and executes in the non-UNIX environment of the Macintosh. Many applications have been developed for the DMD, one of the most widely acclaimed being a multi-window debugger [CARGILL 83]. The close reliance on host support for interface processes means that applications must usually be re-written in order to take advantage of the DMD; the "bolt-on" interface model proposed for MPFW has not been investigated.

2.3.3 Network Windowing Systems and Standards

Vendor-independent windowing systems for UNIX have been the subject of several major research projects in recent years. X windows (MIT), NEWS (Sun Microsystems), and to a lesser extent ANDREW (CMU - [MORRIS 86]) have evolved from their experimental origins and are now in general use.

At one stage, a Macintosh implementation of one of these standards (X windows) was considered as a possible vehicle for the project, but after delays in obtaining the source code this idea was abandoned. NEWS was brought to the attention of the author only at the latter stages of the MPFW's implementation, and its influence was therefore limited. Nevertheless, these are standards against which MPFW should be compared, and the issues raised in their design remain the subject of debate in the field of man-machine interaction.

In retrospect, the disadvantages in porting one of these systems would have been similar to those raised in the previous section - the Macintosh interface model is standardised and carefully integrated with the Macintosh hardware, and user opinion (inferred from USENET discussions) appears to favour many of its design

decisions over those of other schemes. Although details of interface appearence are less heavily embedded in vendor-independent models than in workstation designs, the structural conflicts between the systems - such as event handling, graphics bases and screen management - would have been very difficult to resolve.

2.3.3.1 X

The X Window System™ from Project Athena at MIT [GETTYS 85a,b] has been promoted as a standard for window management under UNIX, notwithstanding recent gains in the popularity of NeWS.

X uses a server process on the workstation to drive the interface hardware (usually a mouse, keyboard and bit-mapped display). Client applications request services through a message passing protocol which runs over standard inter-process communications channels, allowing client processes to run on a different machine from the server. The X server does not contain application-specific code - clients have access to a standard set of library calls from which they must build their own interfaces. Several ready-written client applications are provided with the package, including two windowing shells.

2.3.3.2 NEWS

NEWS™ [SUN 87] is a device-independent user interface management system developed at Sun Microsystems after their in-house SunDew system [GOSLING 85]. The structure of NEWS is similar to that of X, only somewhat more flexible in that it uses the POSTSCRIPT™ language for communication between client and server instead of a non-extensible remote procedure call mechanism. Hence the functions required by the client can be programmed into the server if they are not already present as primitive operations, and this can reduce the amount of traffic considerably as well as simplifying the programming of client. For example, a routine to draw a particular style of window can be loaded into the server, avoiding the need to call each component primitive operation whenever the window must be re-drawn.

™ The X Window System is a tracemark of the Massachusetts Institute of Technology.

™ NeWS is a trademark of Sun Microsystems, Inc.

™ POSTSCRIPT is a trademark of Adobe Systems Inc.

The rôle of multi-threading in user interface management systems - particularly in respect to event synchronisation, resource allocation and server state maintenance - is a current topic of research [TANNER 86, LANTZ 86a,b]. Internally, the NEWS server employs separate threads for each client, and in this respect it is similar to MPFW.

2.3.3.3 JADE

The JADE project was a series of experimental systems developed at the University of Calgary between 1983 and 1986. Part of the project was concerned with the study of user interface management systems in an environment of distributed workstations, and included development of general guidelines for user interfaces [HILL 84,86]. These guidelines were useful for designing the interfaces of §2.5 and for considering the range (and restriction) of capabilities which MPFW should support.

One experimental interface [GREENBERG 85] from the JADE project was considered as a model for the MPFW shell interface; the WATCHER system in Macscrewn (§2.6.2) also bears some similarity to this design.

2.3.4 Macintosh-based Systems

2.3.4.1 UW

The UW windowing package for UNIX [BRUNER 85] has much in common with MPFW, not least because it is designed to run on a Macintosh connected to the host through a serial line. Some of the ideas for the interfaces and for the communications structure of MPFW were guided by the design of UW.

Each window is treated as a terminal emulator, with a choice of text-only or graphics capable emulators. Window selection, sizing and placement are performed locally using the mouse, although remote window creation is allowed and recent versions provide some other remote services (e.g. re-titling of windows) for host programs incorporating UW's library routines.

The mouse may be use to cut and paste text into the terminal input stream, which is useful as a generalised history mechanism. Alternatively, mouse events may be sent to the host as an escape sequence which certain applications (such as EMACS and CHEF) can interpret for cursor positioning.

Terminal functions are invoked through escape sequences. The range and complexity of messages is not great, and the results of missing a character would not be disastrous - input or output being directed to the wrong window would probably be the worst case, and this could usually be detected and corrected by the user. Protocol reliability is therefore not a major issue; visual feedback is sufficient for UW, as it is for a simple terminal.

2.3.4.2 MacNix

MacNix is a recent commercial product from Eurosoft. I know of it only by reputation, and much of what follows is conjecture, but apparently it operates by mounting a section of a UNIX file system as a virtual disk on the Macintosh - effectively, it implements a Network File System client. File operations are serviced by a host process which communicates with the Macintosh over a serial line or local area network.

This approach has much to recommend it, and in many ways finesses the problem of providing a Macintosh-like interface to UNIX: file system manipulation can be performed through the Finder, and standard Macintosh applications (text editors etc) can operate directly on UNIX files. It does nothing for utilities which have no direct Macintosh equivalent, however, and it is restricted by file system incompatibilities such as UNIX links and file modes.

MacNix alone does not address the same problem as MPFW and other UNIX interfaces; a UNIX host makes a rather expensive Macintosh file server, so presumably the MacNix package offers some means of accessing UNIX utilities if only through a terminal emulator. But used in conjunction with a system such as MPFW, MacNix would obviate the need to re-implement functions which can be handled adequately by existing Macintosh applications.

2.4 Targeting of UNIX Utilities for Interface Improvement

In the paper "Interface Design and Multivariate Analysis of UNIX" [Hansen 84], the authors describe a method of measuring:

- which commands and command groups are used most frequently, and
- which commands commonly terminate abnormally, probably indicating incorrect use.

Early in the project I intended to apply this scheme to local usage data, and to this end I taped daily accounting logs from March through May of 1986. It became apparent, however, that such an analysis would involve a major diversion from the project, and so the results of the aforementioned paper were taken as the guidelines for usage patterns.

It seemed reasonable to assume that frequently used commands, especially those which were often used incorrectly or in conjunction with the on-line manual, would be suitable targets for improved interfaces. For interactive commands, a custom window interface might be indicated. Commands with a large number of flag options could be launched with the help of a dialog box (c.f. §2.6.2). File system navigation commands, for which Hansen et al. measured the highest usage frequency, would probably benefit from close integration with the shell interface.

2.5 Proposed Interface Designs

Before implementing MPFW, it was necessary to consider a range of interface designs in order to gauge the facilities which would be required in the core application. Delays in the development of MPFW itself precluded the implementation of all but the terminal emulation sub-applications, although some of the ideas have been incorporated in Macscrewn.

2.5.1 Terminal Tool

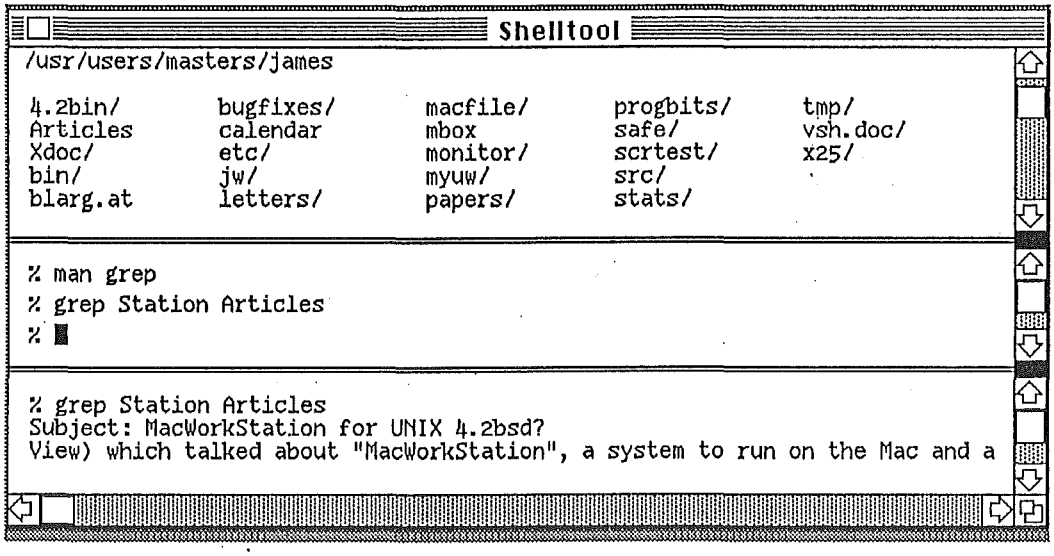
The first sub-application to be invoked when MPFW is launched is a simple terminal emulator, similar to Macscreen. This enables logging in and initialisation of the host multiplexor. Another version can be used once the multiplexing protocol is established, thus providing multi-channel terminal emulation similar to UW. Shell command tools or application-specific interfaces would normally be used in preference to the straight terminal window, except perhaps in the case of interactive applications without custom interfaces. Currently, however, the two terminal tools are the only types which have been implemented.

2.5.2 Shell Interface

The shell tool is a front end for the C-shell [JOY 86a]. Many of the features of the C-shell become less useful in a multi-window environment than they are in a single stream terminal. The history mechanism can be replaced by local editing of the command window, as in the Suntools shell tool. Job control is less important

when processes can each be assigned a window. It is felt, however, that other mechanisms (such as file redirection, automatic pathname generation, variable substitution and control structures) would need to be implemented in any shell which could compete with the C-shell for experienced users. It would work against the nature of this project (and also be very time-consuming) to write a shell of this complexity.

Fig.2.1 Shelltool Window (simulated)



The shell window is divided into three areas (Fig 2.1). The top section contains the name of the current directory and its listing, in the form produced by "ls -F". The user may treewalk the directory structure by double clicking directory names in this list. (Note that ls -F appends '/' to directory names.) The maintenance of these areas is performed as follows:

1) the shell tool generates a "cd <filename>" command - any errors will be reflected in the standard output window.
2) it then issues the command sequence
   "pwd > ttypXX; ls -F > ttypXX &"

where "ttypXX" is a dedicated pseudo-terminal for the top screen area, rather than the standard output pseudo-terminal for the shell. It is unfortunate to require two pseudo-terminal communication channels for this tool, but this is necessary to avoid corruption of output by other processes. The first line of output is written to the top line of the tool; the second goes to the directory list window. In this way the user may continue to work in the command window while the directory areas are updated.

Below these is the command area. Command input and output are kept separate to allow a more effective cut-and-paste mechanism and to maintain a visual reminder of the terminal session history. Command lines are buffered in the shell tool before being passed to the host. This enables local command editing, and also enables the shell to recognise commands for which custom windows or command option dialogs should be invoked. It also allows the trapping of directory changing commands, which are processed as described above, and ensures that automatically generated commands are not interspliced with keyboard commands.

The bottom of the screen is the terminal's standard output and standard error display. This area is normally used only for commands with short output, such as system status enquiries. File listing, manual consultation and editing are directed to separate sub-application windows. Whether command lines are re-echoed in this section is a matter of personal preference; echoing can be controlled by use of the standard STTY command.

In some ways this proposed design may be less robust than an interface to dedicated host shell. Certain commands ("cd", "chdir", "pushd", and "popd", and also commands with custom interfaces) must be caught before they are passed to the shell, and the output of some commands must be posted to specific areas of the window. The commands may be issued erroneously (e.g. with invalid arguments or when the shell is not ready for input) or in ways which may fool a naive interface handler (e.g. issuing "cd" in parentheses, which results in the generation of a *subshell* whose current working directory is changed.)

To counteract these problems, a modified scheme would have the shell launched under the control of a small host monitor application, similar to WATCHER (§2.6.2). This would modify the shell's command responses through standard mechanisms (aliases and environment variables) and thereby supersede local parsing of shell input.

2.5.3 "More" Interface - File and Manual display

This type of window cannot be edited, although it may be useful to allow text to be cut from the display window into the clipboard. The commands MORE (pager) and MAN (on-line documentation through a pager) are trapped by the shell interface, and output for these commands is directed to the appropriate window type.

Each MORE window contains a dialog box (displayed optionally) with check boxes for options such as scrolling style, and buttons for commands such as reverse paging and pattern searching. An edit item in the dialog box is used for pattern specification. The window may also contain a scroll bar which will generate file positioning commands based in MORE's status line if the input is seekable.

When a MORE window is re-sized, a special request is sent to the host monitor to make an ioctl() call setting the pseudo-terminal's size parameters. This facility may also be employed by other tools, but note that few UNIX applications currently take notice of these parameters.

As with the shell tool, this design may prove susceptible to synchronisation problems. One of the design philosophies of MPFW was that no specialised UNIX software (apart from the multiplexor) should be required. In this case, however, a small dedicated file viewing utility would be a great advantage. All that would be required of this utility would be the ability to read specified lines from its standard input file and pass them to the terminal. When its standard input was a non-seekable device rather than a reguar file, the stream could be stored in a temporary file for subsequent seeking. (This method is used in the Suntools CMDTOOL and in the paging utility YAP [Jacobs 85].)

2.5.4 Mail Interface

A basic interface for MAIL [SHOENS 86] consists of a standard terminal emulator with a dialog box. The dialog box contains named buttons for the common mail commands, and check boxes for certain options such as header suppression. This design is similar to the Suntools mail interface, and has been implemented under the WATCHER system of Macscrewn (§2.6.2). A more sophisticated interface might provide separate window sections for displaying the active headers and the contents of the folder directory.

2.5.5 Distributed Editor

The most ambitious interface designed for the project was a distributed screen editor. This editor maintains about 30 lines of text presented in a similar manner to the editors provided with various Macintosh development systems. The host runs a simple line-based editor such as ED [KERNIGHAN 79]. Lines are read from the host

editor's work file and passed to the tool, and are replaced after screen editing when they scroll out of the tool's local buffer.

Pattern matches and global replacements are performed using the facilities of the host editor, with commands issued from a dialog box. Before replacements are performed, all lines in the local editing buffer are returned to the host.

Replacement of all lines is a simple solution to consistency problems. Unfortunately it generates unnecessary traffic when lines are not modified, and it may make the editing sub-application unacceptably slow. Alternative schemes for distributed editing have also been considered. Briefly :-

- downloading the entire file and editing it locally. Using serial communications, this would probably be unacceptably slow when loading and writing the file.
- using Macintosh controls to provide input for a standard display editor, such as VI [JOY 86].
- implementing a Network File System [SUN 86c] client for the Macintosh, thus allowing transparent use of standard Macintosh text editors. (c.f. §2.3.4.2 - MacNix)

2.6 Interface Prototyping

The viability of MPFW's communications protocol and scheduled multitasking remained unproved (to the author's satisfaction) until very late in the project. If the system had foundered on either of these experimental techniques, this thesis would have been seriously undermined. It was therefore considered prudent to implement some of the user interface principles in an alternative environment. A number of small enhancements and one major interface - a variation of Greenberg's workbench dialogs - were incorporated into Macscrewn.

2.6.1 User-Level Customisation in Macscrewn

User-customisation of Macscrewn's interface was considered important for two reasons:

i) to take into account variations in user preference - such things as window positioning, screen blanking delay, and palette appearence

ii) because many of the extensions to Macscrewn were dependent upon the host characteristics (backspace translation etc.) and preferred software packages (e.g. text editor command mapping for the mouse and keypad)

In order to allow multiple invocations of the program from a shared file server, preferences are not recorded in the application file, but users have personal configuration files whose resources override the defaults. These files can be used to launch the program, or loaded subsequently through a menu selection.

The following options can be set through menu selections, and may be saved in a configuration file :-

- delay before a phosphor-protecting moiré display is invoked when the program is idle
- code sent by the 'Backspace' key (erase ^H or delete ^? - reversed when shift-modified)
- code sent from a mouse click (repeated arrow key codes or an absolute positioning code)
- interpretation of option key (editing code generation or high bit setting - 'meta' key)
- verification of incoming file names during file transfer
- use of flow control, which is undesirable with some interactive applications
- use of the 'Enter' key to toggle a mode in which outgoing characters are preceded by appropriate insert/delete sequences to simulate continuous insertion for text editors which lack this capability

Other uses for configuration files include:

- replacement of the function button palette, as described in the following section
- re-mapping of the codes sent by key combinations. This can be used to map arrow and keypad keys for editing functions in interactive applications
- alternative arrangements of the button palette and screen window
- background patterns for the button palette to soften the otherwise stark interface appearance and to differentiate the palette from the screen. (These last two enhancements were incorporated at the suggestion of Mr. Robert Biddle, former supervisor and one of Macscrewn's many long-suffering beta testers.)

Configuration files can also be loaded in response to a command sequence from the host, allowing applications to configure the terminal according to their needs. Alternatively, applications can set many of the parameters with direct command sequences and thereby avoid the need for a separate file for each combination. The use of configuration files to modify existing elements of the user interface for particular applications has no direct parallel in MPFW or existing sub-applications, but has allowed verification of techniques which are used in the resource management of the sub-applications themselves.

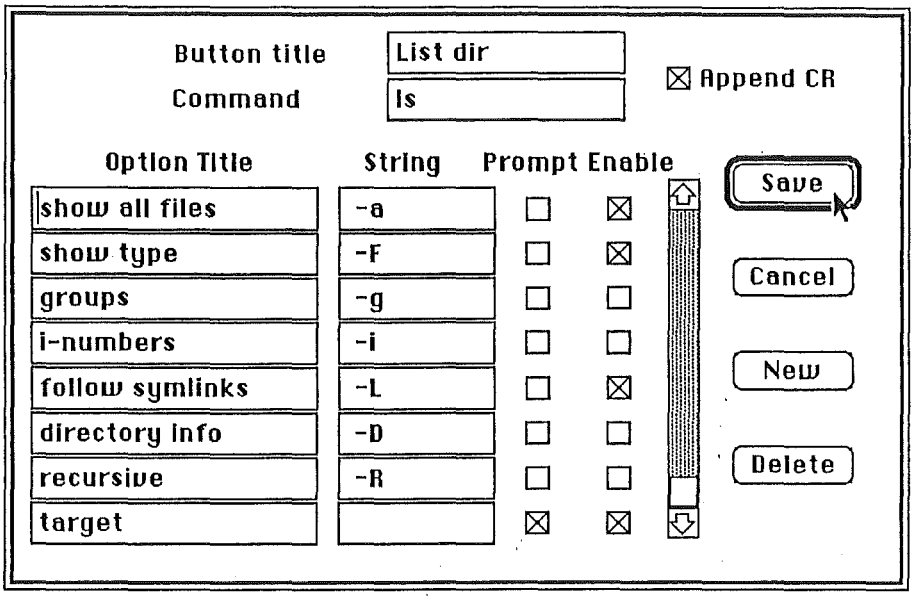
Not all serial lines in the Department operate at the same default baud rate, and it is desirable if Macscrewn can start up at the correct rate. Whereas a given user may work in one of several locations, the Macintoshes themselves tend to remain in one place, so the initial speed should be stored in the machine rather than in the application or its launch parameter file. The non-volatile RAM unit contains a serial line configuration word for just this purpose, but many communications applications appear to ignore it. Macscrewn sets this value whenever the user adjusts the line speed manually, and consults it before initialising the serial driver.

2.6.2 The Watcher System

One experimental interface is the WATCHER system, based on a model proposed by Saul Greenberg in the JADE project [GREENBERG 85]. It provides button palettes for certain UNIX utilities, each containing buttons for commonly used commands.

Macscrewn incorporates an interactive editor for creating and modifying WATCHER interfaces. Interfaces have been implemented for the C-shell, MAIL, RN (news reader) and EMACS (editor) utilities. The buttons are labelled with intelligible command names, and are mapped internally to the appropriate command strings. A dialog listing the command arguments and option flags is associated with each command, and can be accessed by the user to select default command options through check boxes. Fig 2.2 shows this options dialog for the "list directory" button in the C-shell interface. Some commands will require the user to type in arguments, and prompt dialogs for this purpose can be arranged through the options dialog. Once all options have been assembled, the command string is sent to the host as if it had been typed directly by the user.

Fig 2.2 Options Dialog for 'List Directory' Command



As an example, a user might select the "Send mail" button from the C-shell palette. (Fig 2.3) The system puts up dialogs to prompt for the recipient and subject headings, and then sends the assembled command:

```
mail -s <subject> <recipient>  CR
```

This invokes the MAIL utility on the host, which has its own WATCHER interface, and the button palette is therefore replaced with commands pertaining to MAIL. The user starts to type, but at some point decides that the message requires editing. He or she selects the "Edit" button, which sends

```
CR ~v CR
```

MAIL launches the user's specified visual editor - EMACS, in this case - which again requires a new palette. Once the editing is done, the user hits the EMACS "Exit" button, sending

```
CTRL-X CTRL-C
```

The editor terminates and MAIL resumes with its palette restored. (Note that WATCHER does not rely on the use of the palettes for correct operation: the sequence would still have worked if the appropriate sequence had been typed explicitly). Finally the user selects "End letter", which posts the letter, terminates MAIL and returns to the shell with the command

```
CR . CR
```

Fig 2.3 C-shell Command Button Palette

WATCHER is the name of a small UNIX program which strives to keep the button palette and the foreground application matched correctly. Each UNIX command which has a special interface is overridden in the user's search path by a link to WATCHER. When WATCHER is invoked through one of these links, it consults a table which associates the command name with the appropriate configuration file and the full pathname of the UNIX executable. WATCHER then installs the configuration file and spawns the command, adding the file name to the environment list of the new process. When the command terminates or is suspended, WATCHER intercepts the resulting signal and calls for the palette to be replaced from the original configuration file named in the inherited environment; if none exists, the default palette is restored. When a stopped command resumes, WATCHER is woken with the command's process group and restores the palette. In this way, the system can retain the correct interface settings in the face of command sub-execution and BSD-style job control, without requiring a modified shell. Performance is acceptable; the major components of the delay are the loading of the configuration file (~4s for the first invocation only) and the re-drawing of the palette (~1s).

Chapter 3

A Critical Introduction to the Macintosh

3.1 The Macintosh User Interface

The Apple Macintosh user interface - essentially a mouse and bit-mapped display windowing system - has become very popular since the first Macintosh was released in 1984. It differed from most experimental interface systems of the time in that far more attention was given to details of human interaction than to an abstract interface model and regular programming structure: the emphasis was on the implementation of a fixed set of graphical interaction entities - for example windows, dialogs, controls and menus and on providing detailed guidelines for the applications programmer on the use of these entities.

The routines which support the user interface are collectively known as the *toolbox* routines, and are stored in ROM along with the operating system support routines. For documentation purposes these are grouped under *managers*, each of which covers a particular aspect of the user interface. The managers for graphical entities - menus, windows, controls etc. - usually include at least two levels of support, one for system defined templates and one for user defined types. Implementation of user-defined controls, windows etc. is considerably more difficult than using the pre-defined types; this factor has encouraged regularity in a wide range of applications. The major toolbox managers are:

- window manager: concerned with the maintenance of overlapping windows, including their allocation, drawing, positioning and sizing.
- control manager: handles interactive graphical controls - buttons, check boxes, scroll bars and user-defined types.
- dialog manager: supports dialogs, which are specialised windows (usually displayed only briefly) for diagnostic messages and prompts, and for the input of structured information through controls and fill-in text boxes. Dialogs may be *modal*, in which case no other actions can be performed while the dialog is displayed.
- scrap manager: provides a standard mechanism for cutting and pasting sections of a document, and for exchanging such sections between documents and/or applications.

- menu manager: supports the pull-down menus featured in all conforming Macintosh applications.
- font manager: handles the accessibility and display of fonts, including size and typeface (bold, italic etc) transformation.
- event manager: provides a somewhat higher-level interface to the Operating System Event Manager (§3.2.6) for applications.
- TextEdit: is a collection of routines for simplifying basic text formatting and editing
- QuickDraw: is the underlying integer graphics system for all interface drawing, and also for application graphics.

There are also a number of *packages*, which do not reside in ROM but are stored in the *system file* (q.v.) and loaded by the application. Packages include floating point and transcendental mathematics libraries, international configurations for such things as currency, date and number formats, and a file selection package (Standard File).

Extensive collaboration with graphic designers and human factors experts is manifest in the standard interface. Its aesthetic details such as the edge shadowing and title-bar appearance of pre-defined windows play a surprising part in captivating the user in the graphical metaphors, as well as distinguishing objects on the screen. Subjectively, the interaction with the graphical interface entities seems more regular and predictable in most Macintosh applications than in comparable interface environments such as Suntools or IBM personal computer windowing applications.

3.2 The Macintosh Operating System

3.2.1 Hardware

The Macintosh CPU is a Motorola MC68000. Early models came with 128K RAM, and a 64K ROM containing the operating system and toolbox routines. The Macintosh Plus (1986) provided 1M RAM with expansion options to 4M, and the ROM was extended to 128K. Most of the original machines have been upgraded to take advantage of new software, and an entry-level 512K RAM version was also produced. The most recent models (SE and II) have a 256K ROM; neither machine has been evaluated in the course of this project, and the hardware architecture of the Mac II appears sufficiently changed to exclude it from some of the description in

this section. Models up to and including the Macintosh Plus contain the following built-in I/O devices:

- a video display controller, supporting a built-in monochrome screen of 512x384 pixels
- a four-voice sound generator
- the "Integrated Woz Machine", controller for the internal 3½ inch diskette drive; (400K on early machines, 800K since Macintosh Plus)
- a Synertek SY6522 Versatile Interface Adaptor ("VIA" in the text) - interface for a number of devices, including the keyboard, mouse, and the battery-powered clock and non-volatile parameter RAM unit
- a Zilog Z8530 dual Serial Communications Controller ("SCC" in the text), used for RS422 asynchronous communications and the AppleTalk™ local area network

Operating system and toolbox routines in ROM are accessed through unimplemented MC68000 instructions ("A-traps", from the hexadecimal instruction format AXXX). The A-trap vector leads to a trap dispatch routine, which reads the trap word referenced by the return address in the exception stack frame. The lower 8 bits of the trap word give an index into a table of system routine pointers, and this gives the correct address for the routine independent of ROM revisions and system patches. The 64K ROM table uses 15-bit routine offsets with one bit to determine whether the routine resides in RAM or ROM; the 128K ROM uses two tables of 32-bit offsets and must select the appropriate table with an extra bit in the trap word.

3.2.2 Memory Management

The Macintosh does not include a hardware memory management unit, which unfortunately precludes the use of protected address spaces for the multi-tasking system to be discussed in Chapter 5. The operating system also expects all code to execute in privileged (supervisor) mode. The 24-bit address space is divided into four equal areas, with RAM occupying the lowest 4M, ROM the next, and peripheral devices mapped into the upper areas.

The screen and sound generator buffers occupy the highest installed RAM addresses. Below these are alternative buffers and certain sections of code and data (e.g. debuggers, ram disks) which remain resident between applications. Next

comes the application area. In normal single-tasking operation, this space is overwritten each time a new application is launched, and is divided as follows:

- Jump Table
- Application Parameters (32 bytes - mostly obsolete): A5 points to the base of this block
- Application globals (Not used in Megamax C; A4 is used to access strings and global data)
- Stack (A7: stack pointer; A6: frame pointer)
- Application heap

Below the application area is the system heap. This contains various items which must remain intact between application launches - data structures for the file system, the event queue, ROM patches etc. The lowest addresses contain items whose absolute addresses are known to the operating system. These include:

- trap dispatch tables
- flags and temporary storage locations for toolbox and operating system managers, and areas for communicating information from interrupt level device driver routines
- pointers to memory divisions and system data structures
- 12 bytes reserved for use by the application (APPLSCRATCH)
- hardware interrupt vectors

The memory management techniques employed in Macintosh programming apply to the heap, which in a normal application contains most of the code and data structures. The Memory Manager allocates variable sized blocks of memory as the application requests them. Each block contains a header which records its size and status (free, fixed or relocatable), and links it to the next block. Requests may be for a *pointer*, which addresses a fixed block of memory, or for a *handle*, which is an indirect reference to a relocatable block. A handle contains the address of a fixed master pointer, which in turn holds the 24 bit address of the block and eight flag bits - two of these bits indicate whether the block is temporarily locked down and whether its contents may be purged. Relocatable blocks are recommended for most purposes to allow the Memory Manager to coalesce free blocks when subsequent memory requests are made, but the programmer must remember that many system routines can request memory and hence invalidate a dereferenced handle. There are

guidelines and support routines for avoiding heap fragmentation, such as reserving space at the bottom of the heap before requesting a fixed block.

Executable code is also stored in heap blocks. In order for code to be relocatable, all branches must be made relative to the program counter and are hence limited to 16 bits. This restricts the the maximum code segment to 32K; cross segment references are made indirectly through the *Jump Table,* which is located at a known offset from register A5. The *Segment Loader* locks code segments into memory as they are needed, and supplies a routine to free them when the procedures they contain are no longer active.

The Memory Manager provides routines for initialising areas of memory as *heap zones*, each of which behaves as an independent heap. There are also facilities for installing procedures to handle failed memory requests. Further detail may be found in *Inside Macintosh* [APPLE 85,86].

3.2.3 File System

A Macintosh file contains two ordered byte streams, called *forks*. The *resource fork* consists of formatted data structures which are manipulated by the *resource manager* (qv). The data *fork* has no structure imposed by the operating system, and is read and written directly by application programs.

Files are stored on *volumes*. A volume can be any appropriately formatted section of storage medium, for example a $3\frac{1}{2}$ inch floppy diskette, a hard disk partition, or a formatted block of memory set aside as a *ram disk*..

The original Macintosh file system (MFS) is a single level structure, with all files on a given volume referenced from a single index. A multi-level system of *folders* is imposed by the Finder application through the per-volume *Desktop* file. This file is private to the Finder and contains icons and other resources used to maintain the desktop metaphor of its graphical user interface.

With the advent of large-capacity hard disks for Macintosh systems, a true multi-level directory structure became necessary, both for performance reasons and also to make the folder structure visible to other applications. Apple brought out the Hierarchical File System (HFS) in the Macintosh Plus 128K ROM and later models. Under the HFS, files are referenced internally through a B*-tree structured

catalogue file, which also contains some of the interface information previously stored in the Desktop file.

Several sets of standard subroutines make up the programming interface to the file system. Sufficient flexibility is rarely offered by language-specific routines such as the standard Pascal I/O calls and the *stdio* libraries provided with many C development systems. For applications with few complex file manipulation requirements (a surprisingly small set) the high level file manager routines are adequate; these calls provide the basic read, write, open, create and delete functions with a few other utilities such as volume mounting and file locking.

For full functionality, it is necessary to use the low level file system calls, also called *parameter block* calls because they operate directly on a standard system queue element data structure. There are about 50 such calls, of which some 20 have been added for HFS support. The parameter block is a union of 11 major variants, each with 20-30 fields which are used in various combinations by the low-level calls.

3.2.4 Resources

The Resource Manager provides a standard mechanism for storing and accessing formatted data structures in files. Each resource is tagged by a four-character *resource type* which identifies the data format; resource instances are numbered and may also be named. There is a set of routines for creating, loading and manipulating resources, based on the Memory and File Managers. These automate the process of swapping resources to and from disk, so that the in-memory copies can be purged if necessary when they are not in active use.

The system makes heavy use of resources. Code segments and drivers are implemented as resources, as are many structures used by the interface managers - fonts, menus, window templates, control templates and dialog boxes being just a few examples. In fact all elements of the Macintosh toolbox and operating system, apart from the unpatched ROM routines, are stored as resources in the *system file*. The system file is always open, and hence its resources are available to all applications.

One of the chief advantages of resources is that they can be altered without requiring re-compilation of application code. New fonts may be built and installed,

and any application should be able to use them without modification. If the strings displayed in menus, dialogs, window titles etc. are stored as resources, they can be translated for use in a foreign country. There is an interactive application called ResEdit which displays standard resource types in appropriate graphical forms, and can be used to create, move and modify resources very easily.

3.2.5 Drivers and Desk Accessories

In common with most operating systems, the Macintosh requires device drivers to transfer information between external devices and an executing application. These have a synchronous side for the application, comprising open, close, read, write and control calls with semantics similar to the file system, and an interrupt-driven side which communicates with the hardware devices.

The single-threaded nature of the (original) Macintosh operating system was recognised as a considerable limitation. By extending the driver mechanism to handle *desk accessories* it was possible to provide access to a number of small accessory programs from a normal application. There are desk accessories for setting system parameters (Chooser and Control Panel) and manipulating files, as well as calculators, simple editors, and many others. Unfortunately they are difficult to write, since they consist of a few routines responding to external calls, and must maintain state information explicitly between these calls. There are also restrictions imposed by the necessity of sharing their environment with the application - for example they cannot use the jump table. Applications must be aware of the requirements of desk accessories and build in quite complex code to support them.

3.2.6 Events

External events are communicated to applications through the *event queue*. The application calls *getnextevent()* to retrieve the next *event record* from the head of the queue. (Some window events are not stored in the queue and have special priorities, so events may not be in strictly chronological order). Each event record contains information required for identifying and responding to the event, *viz* :

- event type
- event message (specific to type)
- modifiers (state of the modifier keys and mouse button)

- time since startup (in *ticks* - 1/60 second)
- location of the mouse in global co-ordinates

There are 15 defined event types:

Event type	Message field
Null	ignored
key up/down/repeat	key code and character code
mouse up/down	ignored
window to be activated/updated	window pointer
disk inserted	drive number
network event	variable
driver notification	variable
application defined (4 event types)	as specified by the programmer

3.2.7 Application Structure

The Macintosh is essentially a single-tasking machine, allowing only one application to be active at a time, although there are programs such as SWITCHER which load several programs simultaneously and allow the user to alternate between them. The new MULTIFINDER from Apple goes further by displaying windows from different applications, and uses non-preemptive multitasking to allow some background processing.

The first phase of a Macintosh application involves the initialisation of the toolbox managers (a single call for each), the menu bar, and the program's internal structures. The first three menus should conform to a standard format, as they may be required by Desk Accessories:

- the first (apple) menu contains a program information item, followed by a separating line and then the names of the available desk accessories
- the second (**File**) menu should contain certain standard entries such as "open", "close", and "print"
- the third (**Edit**) menu should start with the standard editing commands in order, with their menu-key equivalents

For document oriented applications, if any documents have been selected with the launch they can be retrieved through system routines and should be opened; otherwise the guidelines recommend that a new, untitled document be opened.

During normal execution the program runs a continuous loop, collecting an event record on each pass and processing it according to its contents and the current state of the program. The guidelines recommend that the SystemTask() procedure be called at least every sixtieth of a second to provide time for Desk Accessories, so

each loop pass normally includes such a call. Some event responses may take much longer than this and require extra SystemTask() calls during processing. This is performed automatically by the system routines which handle modal dialogs and track the dragging of windows, controls and menus.

The Macintosh operating system allows applications to schedule asynchronous tasks in response to the completion of I/O calls and at the 60Hz video vertical retrace interrupt. These tasks cannot use the Memory Manager safely, and this indirectly restricts their use of much of the toolbox.

There may also be times when a particular type of event is expected, and it would be cumbersome to return to the main loop to retrieve it. It is possible to request only events of a given type, and this may be advisable in these cases, although care must be taken to avoid locking up the interface if the event is not forthcoming.

3.3 Difficulties in Macintosh Programming

An effective user's model is characterised by regularity in the user interface - the Macintosh is justifiably praised for its achievements in this respect. On the other hand, it is a long process for the programmer to attain a working level of familiarity with the Macintosh, and even experienced programmers find that considerable time must be devoted to slow and frustrating debugging. The Macintosh user interface is implemented through more than 800 system routines, and in any application of reasonable complexity it is necessary to use a surprising proportion of these. Support of menus, desk accessories, window management, cursor handling and countless other "standard" features rely on the programmer's correct use of these calls and also on a continual awareness of operating system requirements which an application must meet. Memory management techniques repeatedly require the programmer to trade off efficiency against safety, and this is a common source of programming error. These problems are compounded by bugs and obscure side effects in ROM routines, documented through some 2000 pages of programming manuals and technical notes. [APPLE 85,86, APPLE TN]

This section looks at some of the limitations of the Macintosh from a programming perspective, paying particular attention to aspects which are addressed in MPFW (Chapter 5).

3.3.1 Low Level of the Toolbox Interface

Much of the regularity of the user interface in Macintosh programs is governed by guidelines and example rather than by automated system routines. Certainly most of the toolbox managers provide standard forms of they entities they control (window frames, buttons, scroll bars, menus etc), but even these require the use of many low-level toolbox routines. A simple modal dialog box with three or four interactive controls may need the support of a few hundred lines of C code in order to apply the user interface guidelines correctly.

The low level of the toolbox interface is particularly tedious during the interpretation of events in the main loop. When the mouse is pressed, the resulting event must go through a very long analysis before the appropriate action can be taken. Large parts of this analysis (and many of the resulting actions) are determined by the user interface guidelines, and are therefore identical in almost every application, but the onus is on the programmer to implement them correctly[1]. Many other systems (e.g. the Commodore Amiga window system) pre-process events to a greater degree, which helps the programmer and avoids inconsistencies between applications. Higher level interpretation and handling of events has been one of the goals of MPFW (§5.5), and the description of its event loop in Appendix V gives some idea of the complexity ordinarily required in a Macintosh application.

Admittedly the original Macintosh ROM was hampered by memory constraints, but by setting the toolbox routines at a low level to maintain flexibility, the designers lost the opportunity to enforce regularity in the interface. The guidelines frequently specify detailed behaviour for an interface entity when it is manipulated by the user. To implement this the programmer must be familiar with many routines from different parts of the toolbox, and apply them in strict sequence following the user's actions. Not only does this make the interface programming long and complex, but a slight misunderstanding of a routine or a deviation from the expected sequence can cause the interaction to break down.

Another liability is the lack of regularity in the routines themselves. There is too little conformity between comparable routines in the toolbox with respect to parameter formats, function results and side effects. Different routines require that

[1] Conforming to the guidelines is of greater importance now that applications may have to share the environment (particularly the screen) under MULTIFINDER. (§3.3.2)

integer coordinates be passed as a structure (normal), a pair of integers in either order, (*setpt(), setorigin()*) or encoded in a 32-bit longword (*menuselect()*). Some parts of the toolbox return error codes; others require a separate function call to detect errors. Side effects, while usually well documented, can be counter-intuitive. The aspects of the interface which are automated by the higher level routines are often unpredictable. The dialog manager, for example, provides four levels of automation for controls in modal dialogs, but does not highlight the default button and fails to re-post the final window deactivation event.

Because of this irregular and low level interface, programmers tend not to acquire a broad mental model of the interface programming principles, but learn to use the toolbox by trial and error. The moderately experienced Macintosh programmer does not expect the first design to produce working interface code; system routines will have been misunderstood, and the code is sufficiently complex to make logical errors inevitable. Software engineering techniques - to the extent that they can be applied to interactive programming - are found to be insufficient, and the test-and-recode development cycle that results can only be as effective as the testing. It may also lead to inadvertent reliance on undocumented features of the toolbox, which helps to explain why so many applications fail with each new release of the operating system.

3.3.2 Operating System Support in a Single-Tasking System

The responsibilities of a Macintosh application extend beyond the job which it is designed to perform. In a single-tasking operating system, the active application has control over the entire machine and must therefore carry out certain tasks which would otherwise be performed invisibly.

The main example of this is the support of desk accessories. The application is required to respond to some events intended for Desk Accessories and pass them back to the operating system, and to call SystemTask() at regular intervals, which requires approximate timing of some operations. It is also expected to set up items in the first three menus (apple, File, and Edit) according to a set format so that they can be shared by Desk Accessories. If these items are not used in the application they should also be enabled and disabled as control passes between the application and the Desk Accessory.

Although the details have yet to be formally documented, it appears that newly designed applications will also be expected to take account of other applications sharing the execution environment under the MultiFinder (§3.2.7). Apparently the call GetNextEvent() - usually the core of the main loop - acts as the process exchange point for the non-preemptive multitasking. It will therefore be necessary for applications to ensure that this call is made regularly (as with SystemTask()) during long processing operations. A higher degree of compliance with the guidelines for window movement will also be desirable to allow regular management of objects on the screen; applications which formerly coped with only one window will now have to allow for others. This leads on from the comments about the low level of window frame manipulation in the previous section.

In the absence of memory protection, each application will have to accept the possibility that other applications may interfere with the system at any time - destruction being the most likely form of interference. Automation of complex and error-prone procedures could reduce this source of programmer anxiety considerably.

The fact that every application is expected to support many facets of the interface and operating system in an identical manner suggests that these facets might be better placed under the control of a single agent; in a pre-emptive multitasking system they could be removed from the visibility of the application, saving inconsistencies and duplication of code. This would also make for a simpler programmer's abstraction of the application program as an independent unit running under the control of the operating system, with the functions of both clearly delineated.

3.3.2.1 Addendum - the Limitations of MultiFinder

Since this chapter was written, MultiFinder has been the subject of some criticism in the USENET forum - the criticism following the lines taken in the above section.

Many non-interactive programs (e.g. file compression and file transfer applications) will lock up the interface for long periods because they fail to call GetNextEvent(). An alternative call, WaitNextEvent(), has been added to improve multitasking reponse under certain conditions, and two of the application defined events have been commandeered for system use. Special resources giving details of

program behaviour are expected in "MultiFinder friendly" applications. Of course, these minor annoyances are to be expected in such a "retro-fitted" system, and one can have nothing but admiration for the programmers who were able to achieve any sort of multitasking with existing applications.

Nevertheless, many contributors are uneasy about these added responsibilities for supporting the operating system environment, particularly the timing predictions needed for regular GetNextEvent() calls, and they have questioned the wisdom of Apple's decision not to pursue pre-emptive multitasking.

3.3.3 File System

In attempting to ensure compatibility between the two Macintosh file systems, HFS and MFS, Apple have created a combined system which is so complicated that is behaviour is neither consistent nor predictable. The subsequent costs in program complexity and reliability have been great. Any program designed for general release is expected to support the old ROM with the MFS, and the new ROM with both HFS and MFS. While backward-compatibility was probably a commercial necessity, it was only partially successful since most applications still required some updating in order to work reliably under the HFS.

The central problem is the identification of files and directories. Under the MFS there are few problems: all file names on a given volume are required to be unique, and currently mounted volumes are identified by a temporary *volume reference number (vrefnum)* . Files can therefore be identified by a <vrefnum, name> pair. Volumes can also be identified by *drive number* or by name, although the latter practice is discouraged on the questionable grounds that identical volume names should be supported. For enumeration purposes, the file system also maintains a close sequence of index numbers for each mounted volume and each file within a volume. Open files are accessed through a *path reference number.*

The HFS introduces a number of new entities, some of which appear to have multiplied without just cause. Directories within a volume are identified by name or by a longword *directory ID* . HFS file and directory index numbers are maintained on a per-directory rather than per-volume basis. Limited backwards compatibility is provided by replacing the volume reference number with a *working directory* reference number (*wdrefnum*): the Finder and Standard File selection package obtain these by opening a working directory for each folder from which documents

are selected. Each working directory structure contains a longword identifying the application responsible for closing the structure when it is no longer needed (usually the Finder).

The high level file system calls can identify unopened files by name and working directory or volume reference number; this is inadequate for many purposes. Using parameter block calls, the target object can be determined by several combinations of the tags mentioned earlier, *viz:*

- by full pathname, from volume down to file name (discouraged)
- by partial pathname with a volume or working directory reference number (referring to the root of the path)
- by partial pathname with a directory ID (referring to the root of the path) and a working directory or volume reference number specifying the appropriate volume
- by a combination of directory specification (any of the above) and index number
- by the above methods, only substituting the default directory or default volume for a specified directory or volume reference number

Herein lies the chaos. The rules for resolving these combinations suggest discovery and back-patching more than deliberate design. Different calls use different search methods according to whether the various fields are null, positive or negative; there are few general rules and many calls have their own irregular schemes which may be documented inadequately or inaccurately.

Worse, the file system attempts to "correct" certain calls by means of the *Poor Man's Search Path (PMSP).* The PMSP is a list of alternative directories to search when a file is not found, including the current default directory and the directory containing the System File. If a file system call is used to check for the existence of a file, the resulting information can be misleading. One of several unfortunate side-effects is that if a program wishes to open a new resource file for writing, and the given file name appears elsewhere in the PMSP, the check for an existing file will pick up the old file. Instead of failing and thereby prompting the program to create a new file, the call results in corruption of the wrong file. Incredibly, the file deletion call also uses the PMSP. Working around this behaviour can be very time-consuming, requiring such ruses as always attempting to create a resource file before opening it for writing. It is possible to disable the PMSP in some calls by

specifying an impossible volume reference number, although it is doubtful whether this feature was intentional.

In short, the Macintosh file system lacks both simplicity and regularity. The high level calls provide insufficient functionality for most applications, and the complexity of the low-level calls is absurd. The irregularity of the file identification mechanisms makes it impossible to form a reliable abstract model of the file system. Many sections of code involving file operations must continually check the both the type of the subject volume and the ROM version, and provide alternative methods for each combination. This is the greatest irony, given that it was Apple's attempt at backward compatibility which precluded any hope of an elegant file system; it would have been preferable if Apple had phased out the MFS by providing temporary routines in the new system file, rather than supporting two very different file systems on the same machine.

3.4 Macintosh Development Environments

3.4.1 Procedural Programming Languages

It appears that the most common programming languages used for Macintosh application development are C and Pascal[1]. Pascal is the official language used in technical documentation [APPLE 85,86] and the stack format for system routines is based on Pascal conventions. The use of Pascal for "serious" development, as opposed to teaching, has been questioned on many grounds, particularly its simplistic and over-restrictive type semantics (e.g. array passing) and its lack of support for system-level programming ([TUCKER 86] and others). However, versions of Pascal designed for the Macintosh are extended to allow strings, arrays, and pointers to be handled in a more general manner, and they support separate compilation. Such rule-bending is necessary given the amount of system interaction required in Macintosh application programming.

Both C and Pascal have their proponents, although C appears more prevalent amongst the UNIX-oriented USENET community. Some C aficionados complain that Pascal tends to be verbose, so that the content of a program can be lost in a morass of keywords, and that its declaration order, with major routines forced to the bottom of the code, makes large programs difficult to read. Others claim that Pascal's

[1] Inferred from user magazines and USENET discussion groups.

strong typing and paucity of low-level operators are a disadvantage because the operating system interface requires considerable low-level programming (e.g. function results with fields encoded in a single integer, structures containing arbitrary pointer types etc). The C addressing model and operators are particularly well suited to the MC68000, which may make it more efficient for some purposes.

On the other hand, there are difficulties with the system routine interface, which expects Pascal stack format and string representation conventions. Pascal also provides greater protection against semantic errors, although some of the limitations of C in this respect can be ameliorated if the LINT [JOHNSON 79] source code verifier is available.

Modula-2 [WIRTH 82] can perhaps be seen as the natural successor to Pascal for professional development, and is gaining favour as a programming language for the Macintosh. At the time when languages were being evaluated for the project, however, there were no sufficiently well-developed Macintosh Modula-2 environments for consideration.

3.4.2 Toolbox and OS Extension Systems

Code libraries and skeletal systems have been provided as aids for Macintosh programming in standard structural languages. Many C systems emulate routines from the UNIX C library, although these are ill-suited to the Macintosh and are mostly used for porting software which makes little use of the standard user interface. The libraries also include *glue routines* to convert argument formats before calling the system traps. There are some code libraries which give higher level access to the user interface, such as the TML Extenders library for their Pascal compiler. Skeletal systems give the overall structure of a Macintosh program, including such things as desk accessory handling and event interpretation, and leave the programmer to add the application-dependent code.

Most of these packages have appeared during the course of this thesis project, and they address some of the problems discussed in §3.3. The MPFW library (Chapter 5) provides similar facilities, although it may be less comprehensive than some of the commercial systems.

3.4.3 Object Oriented Systems

Object orientation has become popular as a programming paradigm for user interface systems in recent years, especially since many interfaces are derived to some extent from the Smalltalk environment [GOLDBERG 84]. It offers the advantages of *modular decomposition* of the components of the user interface, a subject to receive further attention in §5.2. Attribute inheritance in an hierarchy of interaction devices (window objects, control objects, etc) also imposes a regularity which is painfully absent from the design of the Macintosh toolbox.

There are several Smalltalk implementations for the Macintosh, with one recently announced version replacing the standard interface with Macintosh conventions[1]. These environments, like lisp derivatives such as MacScheme™, have high storage overheads and are usually regarded as more suitable for prototyping than the production of commercial software.

Object Pascal adds object oriented structures to standard Pascal. It is the basis of the MacApp, a development system which offers a much higher level interface for the Macintosh toolbox and overcomes many of the deficiencies outlined in §3.3. While it is acclaimed as a prototyping system, however, the programs it produces are reputed to be somewhat slower than those of other systems[2], and it requires a heavy learning investment by prospective users[3] .

Object orientation has also been introduced to C in the Objective C and C++ extensions. They have yet to be incorporated in a Macintosh programming system, but an efficient object orientation scheme combined with the popularity of C on the Macintosh will make these systems influential in future application development.

3.4.4 Choice of Development System for the Project

Because MPFW required some low-level modifications to the Macintosh operating system, and because the communications protocols were to require parallel code on the Macintosh and the UNIX host, C appeared to be the most suitable language for the project.

---

[1] MacSmalltalk from Apple.

™ MacScheme is a trademark of Semantic Microsystems, Inc.

[2] L. D'Oliveiro, in response to my USENET enquiry Jan. 1988

[3] Larry Rosenstein, MacApp group, Apple Computer - USENET article

At the start of the project only one C compiler - Megamax™ C - was available through the Computer Science Department. LightSpeed™ C was purchased several months later, but I decided not to convert MPFW to the newer programming environment, partly because of the work involved, and partly because LightSpeed was less suited to the project than Megamax.

The Megamax compiler does not produce particularly efficient code - switch statements, for example, are implemented by repeated comparison. The Megamax compiler has also dropped out of use because it was producing programs which use a reserved low-memory location to save the contents of the global base register A4. A recent update to the Macintosh operating system causes these programs to fail. However, by patching the binary of the compiler and the system library I have corrected this problem and can now use the compiler to produce programs which are compatible with the new system files.

The adherence to Megamax is largely due to its open design. The Megamax object file format is well documented, which made writing the pre-linker for MPFW sub-applications (§5.6) much easier, and the compiler also provides an in-line assembler which proved necessary for process implementation in MPFW. LightSpeed is at a disadvantage in both these respects. The components of the Megamax system - the editor, compiler and linker, with a number of support programs including a librarian and code optimiser - are all implemented as separate applications, and the development cycle can be automated through a shell language interpreter called BatchX. This gives flexibility to the system, allowing replacement of components and extensions to the system by writing small C programs to call from BatchX. A distributed development environment based on these capabilities forms part of the subject of the following chapter. Such a design would not have been possible with LightSpeed, which is a monolithic and non-extensible system.

™ Megamax C development system V1.0 ©1985; converted later to V3.1 ©1987

™ LightSpeed C © THINK Technologies 1986

Chapter 4

Development Support

4.1 Distributed Development Environment

Several major Macintosh programming projects have been undertaken in the course of this thesis. Each has required a large number of source files (varying from about 15 to over 40 in the case of the shared project library), and some projects have required complex compiling and linking sequences. Some form of automatic compilation control with source code dependency management was necessary under these conditions. There were also reasons for keeping copies of the source code on the host - as a backup location, and to allow access to UNIX tools. This section describes a custom-built development environment with components on the Macintosh and the UNIX host, and also the modifications to Macscrewn which enabled it to act as the communications agent in this system.

4.1.1 Remote Application Launching

One method of automating long Macintosh tasks is to allow scripts executing on the host to invoke a sequence of Macintosh applications. This requires a facility for sub-launching an application (with arguments) from the terminal emulator in response to a host command sequence.

The mechanics of sub-launching are reasonably well known - the Megamax system provides a routine for launching an application with arguments, and Macscrewn uses the same method as BatchX to regain control after the application has finished: it installs itself as the Finder before each launch, and restores the original Finder when it is restarted. Macscrewn cannot retain state information between invocations, but the loss of screen contents is of little importance since this feature is primarily designed for non-interactive operations.

One requirement of this scheme is to delay the host script interpreter while the sub-launched application is active. It is also necessary to prevent the interpreter from exiting in response to the hangup signal generated by the driver when the terminal line drops. To accomplish this, a small program called MACEXEC is used to communicate the launch request and then hold the line while the sub-launched

application is active. When Macscrewn or another terminal emulator regains control, MACEXEC exits and allows the script interpreter (or the interactive user) to continue.

4.1.2 Command Driven File Transfer

Macscrewn uses the XMODEM-based MACPUT and MACGET utilities for transferring files between the Macintosh and the UNIX host. As supplied, these utilities require manual control of each file transfer, which is unduly laborious in a distributed development environment where several dozen files are regularly loaded from one machine to the other. The protocol has therefore been extended to support script driven transfer and uploading of Macintosh directory sub-trees. Other features have been added to maintain consistency between text files which must be used on both machines. The options provided require modification of the host utilities, but the utilities remain compatible with the standard protocol.

When copies of a text file are stored on both machines, and either copy may be modified, MACPUT and MACGET can be invoked with a *conditional transfer* option. In this mode, the modification times of the files are compared and the transfer proceeds only if the source file has been modified more recently than the destination file. If the modification times are identical, the transfer is deemed unnecessary and is silently ignored, but if the destination file is newer than the source file then the user is alerted to the possible conflict and given the opportunity to continue or abort the transfer.

Several UNIX shell scripts for multiple file transfer have been used throughout the project. These use the non-interactive options of MACPUT and MACGET. By default, they treat the files as text files and use conditional transfer, although these options can be disabled through command line flags. Below are some of the most commonly used scripts:

- PUTSOME: takes a list of UNIX files and downloads them to a specified Macintosh volume or directory.
- GETSOME: uploads listed files from the specified Macintosh volume or directory to the current working UNIX directory.
- PUTPROJ/GETPROJ: like PUTSOME/GETSOME, but the files and target volume are taken from a *project file list* in the current host directory. Normally used for backing up all source files in a particular programming project.

- PGPROJ: works through the local project file list, transferring files in either direction to ensure that the most recent version of each file is present in the project directory on both machines. The compilation control system (§4.1.3) calls this script before it calculates dependencies or invokes any other operations on the project sources.

The existence and modification date of a Macintosh file can be reported through Macscrewn using a small host program called MTIME. This program can be used in conjunction with MACEXEC (§4.1.1) to control conditional compilations directly from a UNIX shell or MAKE script, instead of using a command interpreter such as BatchX. This method is somewhat slower, but is sometimes necessary as BatchX cannot process very large scripts.

4.1.3 Compilation Control

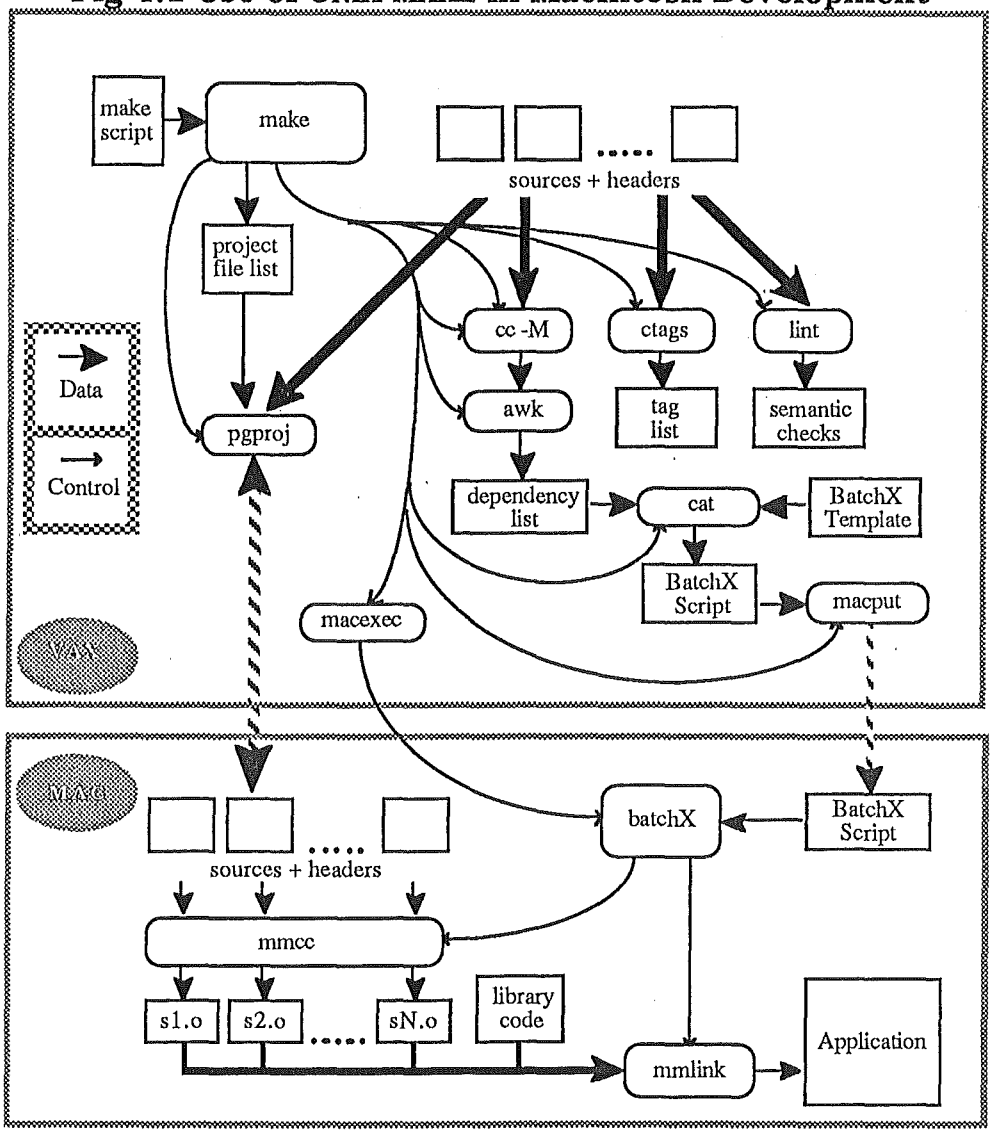
The Megamax BatchX command script interpreter does not have the power of the UNIX shells or the MAKE utility [FELDMAN 79]. Nevertheless it includes sufficient constructs to implement conditional compilation sequences, albeit in a somewhat long-winded manner. Relevant primitives include -

- one dimensional word list operations (creation, assignment, indexing, expansion)
- character string operators
- file existence and modification time comparison
- application sub-launching, with arguments
- condition testing and looping constructs (if-then-else, while (condition), foreach (element in list))
- arithmetic operators

Apart from delayed variable/list expansion, the major omission seemed to be an ability to manipulate files. After the Department acquired a file server, it was possible to access all components of the development system without disk swapping or an external disk drive; in order to avoid anti-social levels of network traffic, however, the scripts required some method of copying the compiler and system header files into a RamDisk when several files were to be compiled. It was also desirable for object files to be moved to a common location in some projects, and for the scripts to be able to record information in a file. Some additional small utility applications were written to extend BatchX's capabilities in these areas:

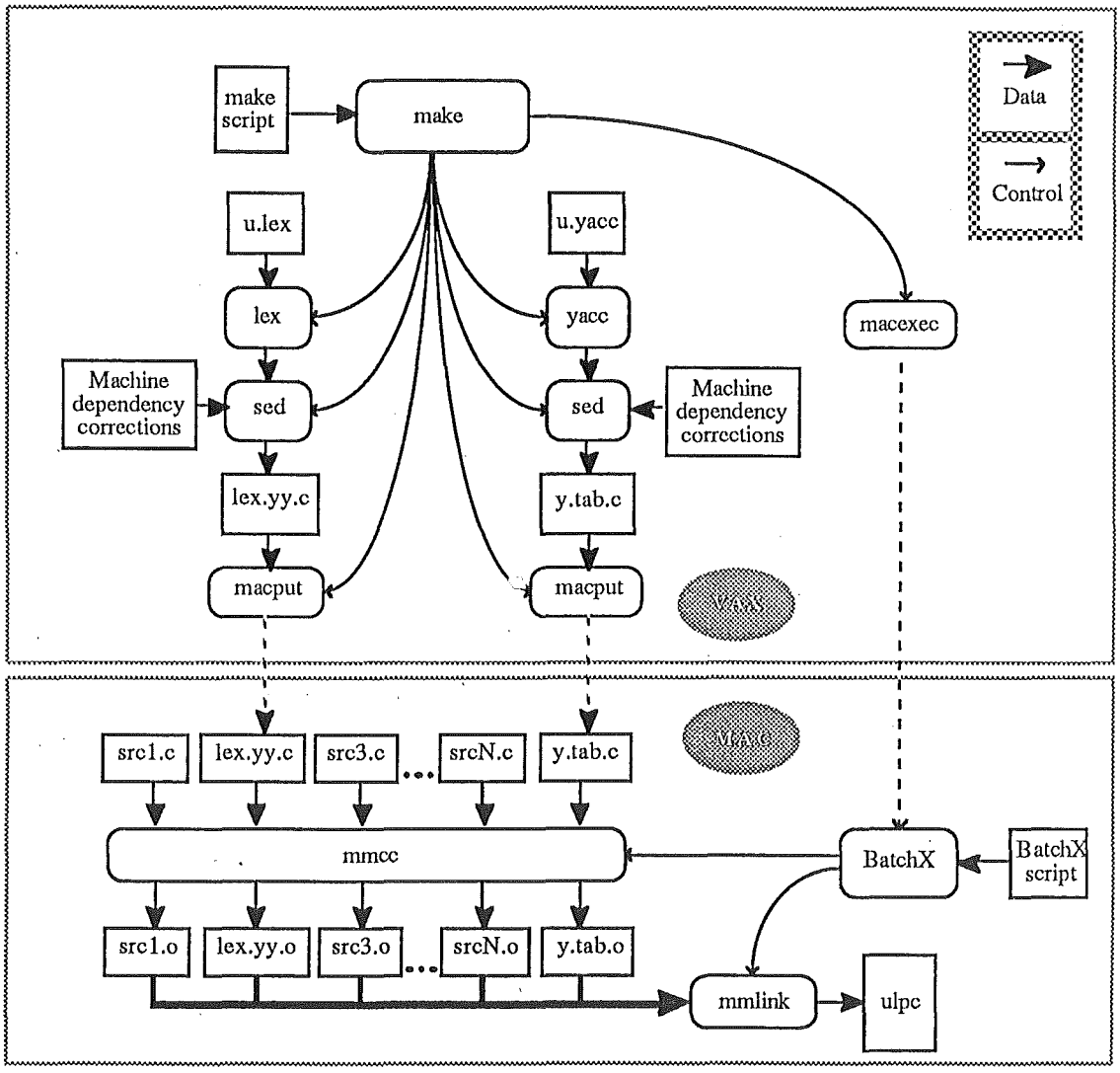
- CP: copies files or directories to a specified volume or directory
- MV: moves files or directories to a specified directory within the same volume
- ECHO: writes its arguments to the file named as its final argument
- APPEND: like ECHO, except that it appends to the file rather than overwriting it

Fig 4.1 Use of UNIX MAKE in Macintosh Development



The chosen method of controlling compilation sequences has been to create dependency lists on the UNIX host from the Megamax C source and header files, and to incorporate these lists in BatchX scripts to be run on the Macintosh (Fig 4.1). It was considered desirable to allow compilations to remain unattended, given that some of the sequences required of the order of half an hour in the worst case. In the absence of programming errors, the project management scheme described can perform a complete re-compilation in response to a single typed command.

Fig. 4.2 Compilation Sequence for ULPL Compiler



This approach requires the maintenance of consistent copies of the files on both machines, but this can hardly be viewed as a disadvantage since it promotes continual update of at least one backup copy of the source. It also allows the programmer to make use of UNIX programming tools which are not available on the Macintosh, such as the CTAGS routine indexing utility and the LINT source code verifier. Many UNIX tools were employed in the development of the Macintosh projects, usually to take advantage of more powerful text processing facilities than were available on the Macintosh - e.g. script-driven regular expression searching and replacement through multiple files. These tools were also useful for processing large quantities of debugging output, such as was produced during the testing of ULPL's code generator (Appendix IV).

Development of the ULPL compiler was further complicated by the fact that two components of the compilation sequence - LEX [LESK 79] and YACC [JOHNSON 75] - were available only on the host. During the early testing of the parser, the program was developed wholly under UNIX so that YACC's debugging facilities could be used. Fig 4.2 outlines the MAKEFILE control which was used for the compiler after it was transferred to the Macintosh. The versions of LEX and YACC supplied with 4.2 BSD UNIX for the VAX make some machine dependent assumptions - 32 bit integers, 16 bit short integers, and a lenient interpretation of the C union tag rules. These are unjustified for Megamax C, and some post-processing of the generated code was necessary. Some of the other projects had UNIX tools built into the compilation sequence in a similar fashion - one example of this was the use of the AWK [AHO 79] text processing language to compile C initialisations into Megamax object code for a prototype MPFW sub-application.

4.1.4 Communications Monitoring

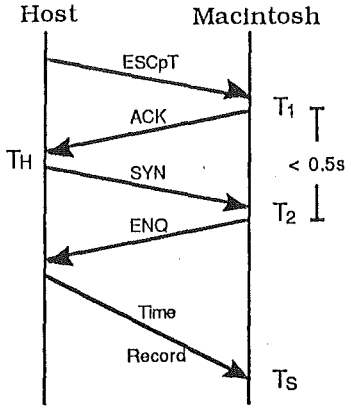
When developing communications software, it is often desirable to be able to view and/or record all characters transmitted by the host, without interpretation of control characters. Macscrewn incorporates its own eight bit font, with symbolic representation of control characters and macrons to mark characters with the high bit set. These are displayed when the *Monitor Mode* option is selected from the menu. If the terminal is set to record incoming characters in this mode, they are recorded verbatim without the usual bit-stripping and carriage return translation. This feature has been of considerable use in debugging the various protocols employed in the project, including the MPFW communications module (Chapter 6) and the extended XMODEM implementation in Macscrewn itself.

4.1.5 Clock Synchronisation

Because file modification times are used by both the project management systems (MAKE and BatchX), it is important that the clocks on the hosts and the Macintosh should be synchronised, preferably to within a fraction of a second. For this reason, Macscrewn supports a protocol which adjusts the Macintosh internal clock through the host program MACSYNC. MACSYNC is normally invoked during the login sequence, and can also be used to identify the terminal before setting host parameters. For improved accuracy, the protocol attempts to counteract communications delays (slow line speed, heavily loaded host etc.) which would

otherwise cause the Macintosh clock to be set slow with respect to the host. The sequence is as follows:

Fig 4.3 Clock Synchronisation Sequence



i. Host sends the function identification sequence "ESC p T".

ii. Macscrewn notes the current Macintosh time T_1, and responds with the character "ACK".

iii. If no response arrives at the host within a given timeout, MACSYNC assumes that it is not talking to Macscrewn and returns a failure code. Otherwise it records the current host time T_H and sends a "SYN" character.

iv. Upon receipt of the "SYN", Macscrewn notes the current Macintosh time T_2. If T_1 and T_2 differ by less than 0.5s, reasonable bounds of reference for T_H have been established, and it responds with "ENQ". If the difference is too great, (i.e. if the "SYN" times out), it resets T_1 and sends a negative acknowledgement ("NAK"), and the process returns to step 3. Three failures are allowed, after which the synchronisation is abandoned with a "CAN".

v. Host transmits a 128 byte XMODEM block containing T_H in a UNIX *time_t* structure.

vi. Macscrewn receives the block and converts it into a local time format. It notes the current local time T_s and sets the internal clock to T_M, calculated thus:

$$T_M = T_H + T_s \frac{T_1 + T_2}{2}$$

4.2 Macintosh Library Extensions

With several Macintosh projects under simultaneous development, it was seen as desirable for them to share as much code as possible. This applied particularly in cases where routines had to be developed in response to the problems noted in Chapter 3. An effort was made to ensure that the routines were sufficiently general

to allow their use in the different programs. The Megamax C development system includes an object module library archiver, and this was used to maintain the shared routines. Appendix II contains documentation for the library routines.

4.2.1 File Manipulation Library

The file transfer facilities described in §4.1.2 and the resource file access required for the WATCHER system (§2.6.2) require reliable schemes for manipulating Macintosh files in response to commands sent from the remote host. Examples of required functionality include:

- access to files using full or partial pathnames which are stored in the host environment
- working directory creation and manipulation - for use with C *stdio* routines and for application sub-launching
- secure creation and opening of resource files, circumventing the errors induced by the default use of the Poor Man's Search Path
- selection of files by type/author combination
- path stripping for MACGET header generation
- volume and directory indexing for directory copying
- HFS/MFS transparency

The high level routines used in applications which access all files through the Standard File Package are inadequate for these purposes, and so Macscrewn and other file manipulating programs are forced into the quagmire of the parameter block routines (§3.2.3, §3.3.3).

Correct file manipulation code at this level is too complex to attempt ad hoc methods in each case. Instead, a new library of file handling routines has been developed to combat the file system problems outlined in Chapter 3. MFS/HFS integration and work-arounds for low-level routine inconsistencies such as the Poor Man's Search Path are built into the library.

The core routine is *makefid()*, which accepts an arbitrary combination of file, volume, or directory identifiers and returns a file identification structure (*fileid*) containing all information required for operating on the given file system object (if it can be identified uniquely and is of the requested type; otherwise it returns NULL). If an object with an unrecognised volume name is specified, *makefid()* will post a dialog box requesting insertion of the named disk - with an option to cancel, of

course! *makefid()* can also be passed flags for creating and or opening the object, based upon the capabilities of the UNIX file system calls (open for reading / writing / both, create if the object does not already exist, truncate on open etc.) with extensions appropriate for the Macintosh (open data fork / resource fork / raw resource fork / working directory, allow shared read/write access). Some measure of the Macintosh file system is revealed in the fact that *makefid()* and its support routines require over 1000 lines of compact C - just to support identification, creation and opening of standard file system objects.

The library also includes:

- a volume/directory indexing module similar to *scandir()* in UNIX
- simplified read and write calls
- conversion routines between *fileid* and Standard File Reply structures
- a file system error interpretation routine
- routines for integration of C *stdio* packages with the *fileid* system

Further details are given in the library descriptions of Appendix II.

4.2.2 Modal Dialog Management

Macscrewn and other programs in the project use a wide range of modal dialogs, each containing a variety of controls and other interface objects. As I mentioned in Chapter 3, modal dialog handling routines usually share a very similar structure but are still laborious to write. The criticism applied throughout the toolbox bears repeating: some automation is provided, but not enough to save the programmer from repetitive implementation of the guidelines in each instance.

A Standard Dialog package was developed for the project library, and this has been found to reduce the difficulty of modal dialog programming while still retaining sufficient flexibility for most purposes. The caller sets up an array of variant records which specify the type, the initial state and the initial value of each item in the dialog. This array is passed by reference to the Standard Dialog routine, which displays and updates the dialog in response to the user's actions, and records changes to item values in the array. Specialised actions which must be performed while the dialog is active can be implemented by replacing Standard Dialog's event filter function, or by specifying action procedures for appropriate items. When the user selects an item which is flagged as a terminator, the dialog is removed and the

terminating item number is returned to the caller. Changes resulting from the dialog interaction are available from the item array.

Because of constraints on the use of *ModalDialog()* imposed by the multitasking nature of MPFW, the Standard Dialog package could not be ported directly to MPFW. A similar package using the simulated modal dialog facilities of MPFW would nevertheless be possible. The Standard Dialog routines have been used in many small support applications for the project.

The interactive creation of dialog boxes is also featured in Macscrewn (§2.6.2). The toolbox does little to support adding and editing items in a visible dialog, nor does it provide for the creation of an item list resource from an active dialog description. The documentation contains only oblique references to these operations, which require interpretation and careful alteration of the underlying data structures. Generalised routines developed for Macscrewn in these areas have been installed in the common project library.

Chapter 5

MPFW - A Framework for Macintosh Development

5.1 Overview of MPFW

Chapter 3 outlined what I see as the major deficiencies in the Macintosh operating system. This chapter describes MPFW, a system which attempts to rectify some of these deficiencies through a number of features including:

- provision of scheduled pre-emptive multitasking to allow simultaneous execution of independent sub-applications
- simplification of the programming interface to the toolbox and operating system through shared library calls
- partial automation of window management and operating system support by system processes

MPFW is not a replacement but an adjuct for the Macintosh operating system. Its status is that of an application running under the normal OS and toolbox environment, although certain modifications to interrupt-level routines are required. These preclude the use of the current version of MPFW with the MC68020 in the Macintosh II, and may result in other incompatibilites with future versions of the operating system. Further details of the multitasking implementation are given in Appendix III.

Sub-applications are independently compiled programs which can run simultaneously as separate processes under MPFW. They reside in separate files, allowing them to hold independent resources and to be stored, modified or moved in isolation from MPFW, although when multiple invocations of the same unit exist they share code and resources. The structure of a sub-application is similar to that of an ordinary Macintosh application, but much of the operating system and interface support is performed by MPFW.

Sub-applications communicate with MPFW by a combination of "system" calls and shared data structures. The "system" calls and the toolbox and OS glue routines reside in MPFW and need not be duplicated in each application. There are also about 50 shared library routines for assisting such things as file and dialog management. These encapsulate large sections of code which automate common

operations and work around some of the obscure bugs and inconsistencies in the ROM. The entry points for these routines are resolved in the sub-applications by a pre-linker which relies on the original load map for MPFW. MPFW does not itself require relinking when a new sub-application is created, but the linking process currently relies on the Megamax C object file format and sub-applications must therefore be developed with this system.

Many of the features of MPFW arise from its intended use in a distributed environment, particularly as a user interface agent for a UNIX host. Facilities for external communications are closely embedded in the process structure, and the current system supports the communications protocol described in Chapter 6.

Internally, MPFW is structured as a group of closely intergrated processes, or *threads*, compiled as a single C application and sharing a common global scope. These threads have the same status as the sub-application processes, but do not go through the process of loading external code and resetting the global data area. Eight process priority levels are provided, with priority given (in descending order) to system threads, the owner of the frontmost window and interactive sub-applications. Process synchronisation and resource protection are achieved through semaphores, although for efficiency some routines simply set a process exchange lock to avoid transfer during critical operations. This lock is set automatically during trap calls, because many system routines are not re-entrant.

5.2 Advantages of (Pre-Emptive) Multitasking

The major advantage of any form of multitasking is that it allows (or at least simplifies) the simultaneous use of a range of independently-developed modules, each providing a particular service for the user. In its original form, the Macintosh did offer a limited form of concurrency through desk accessories, so it may be worthwhile providing some definitions for the terms used in this thesis:

- *multitasking* is characterised by the concurrent execution of several tasks, each of which maintains a separate stack context and need offer no explicit support for control transfer apart from calling a scheduling routine at appropriate intervals
- in *pre-emptive* multitasking, control transfer is performed by an interrupt-level routine and is invisible to the task to the extent that no explicit support is required.

By these definitions, desk accessories do not qualify as multitasking entities because they do not maintain separate stacks and can only be called through a fixed set of entry points; all context information must be saved explicitly between calls. The main reason for their exclusion is that the term "multitasking" is to be used in contrast to this mode of operation.

The arguments for multitasking as an aid for the provision of multiple user services are well known. This section will therefore concentrate on multitasking as a programming device, considering the advantages of independent processes and of closely cooperating processes or *threads*.

A multi-threaded system forces the programmer to delimit each task and define explicitly the interfaces between each module. This regularity of the programming interface is especially necessary in an environment where modules are developed in isolation (e.g. MPFW sub-applications). Multitasking hides control transfer from the programmer, making the module appear as a self-contained unit communicating only through specified system calls. This view makes the program structure easier to comprehend: its areas of responsibility can be clearly defined and limited to the task it sets out to perform; its co-existence with other tasks should be hidden by the operating system as far as is possible. Where processes are cooperating for a common purpose, however, the greater motivation for multitasking is the promotion of *modular decomposition* of the work. In the following paragraphs, the terms *vertical* and *horizontal* partitioning are used in reference to two forms of modular decomposition.

Horizontal partitioning is used to implement layered systems, for example communications handlers. Each protocol layer is implemented as a separate process and can be modelled as a continuous operation, taking a unit of information from one layer, processing it according to the current protocol state, and passing it up or down to the next layer. In this way the agent responsible for maintaining a given layer is clearly delineated and can be programmed in a manner which reflects its independence. Also, if the process communications interface is suitably defined, it should be easier to replace individual layers when software and hardware requirements change. Process synchronisation is needed between the layers, but this is often simpler and more efficient than the ad-hoc flow of control which appears in single-threaded layered programs.

The UNIX kernel is not multi-threaded internally, so all processing (apart from interrupt handling) occurs in the context of user processes. Horizontal partitioning through normal processes is nevertheless one of the key strengths of UNIX, occurring in pipelines, window managers and a large number of server programs. Applying the philosophy of conventional UNIX interfaces to the graphical interface, horizontal partitioning can provide a further advantage: configurability. Different interfaces may be required for the same underlying application, according to the available hardware and the sophistication of the user. Graphical interfaces will obviously be inappropriate where output is to be recorded or passed to another application for further processing. Where the user interface is closely embedded in the application, as in existing Macintosh applications, it is almost impossible to make use of the application for non-interactive purposes.

Vertical partitioning promotes the independence of tasks executing at the same level. Such tasks may be running different code, as in the case of system processes and diverse applications executing simultaneously, or they may be clones of the same module which handles, for example, a single channel of a multiplexing communications system or a window of a text editor. In this way the operating system takes over the problems and of allocating and maintaining multiple data structures and of saving states between servicing each unit, and fewer unintentional interdependencies will be introduced. The intrinsic interdependencies of the cooperating tasks can be handled in a regular fashion by message passing, either directly or through a lower level process - in our examples, the next communications layer or the window manager. This implies both horizontal and vertical partitioning.

The topic of multitasking received much attention at a recent ACM SIGGRAPH workshop on user interface systems. The report of one group [LANTZ 87b] suggested four major motivations for using multiple communicating processes in a software system. The first was that of modular decomposition, which corresponds to the advantages noted above. The second motivation was categorised as *protection* - the reduction of side effects through information hiding. Protection is greatly enchanced if processes can be given disjoint address spaces; unfortunately this is not possible on the Macintosh, and for reasons of efficiency MPFW processes sometimes compromise this advantage further by communicating through shared data structures. The remaining motivations (distribution and performance) relate to

the use of multiple processors, which is inapplicable to this project except in the context of separating the functions of the UNIX host from those of the Macintosh interface.

The paper contends that most of the advantages of modular decomposition can be obtained through modern data abstraction languages rather than processes. Given that the other stated advantages are undermined in the Macintosh environment, this seems to question the viability of processes as the basis of MPFW. However, the authors admit that such languages do not solve the problems of systems which handle multiple threads of control, and extoll lightweight processes (same processor and shared address space) as a programming mechanism for these situations. Another report from the workshop [LANTZ 87a] notes the importance of full-blown processes (independent operating system entities, preferably with disjoint address spaces) to promote configurability and avoid monolithic applications. These papers recommend the adoption of both types of process in a model which has much in common with MPFW.

Nevertheless, pre-emptively scheduled processes are not strictly necessary for MPFW. In the original design, each sub-application was a collection of event handling routines sharing a single stack with the core application. This would have forced the sub-applications to store considerable amounts of context information between events, resulting in an awkward programming stucture - a disadvantage shared by Macintosh desk accessories. Secondly, it would be very difficult for sub-applications to complete long processing tasks without disabling the user interface; with pre-emptive multitasking, MPFW is able to respond quickly to high priority events such as abort keystrokes and window selections - unless the sub-appication explicitly disables such interruptions.

Alternatively, the sub-applications could have been implemented as co-routines, relinquishing control explicitly rather than at interrupt level. Burdening the sub-application programmer with scheduling considerations would have destroyed the illusion of the independent sub-application, and ran counter to the aim of simplifying Macintosh programming. It would also have taken away the chance to tune the system and adjust relative process priorities in favour of interactive sub-applications. Against this I had to balance the difficulties of implementing scheduled multitasking, and the knowledge that certain toolbox sequences would have to be

completed with process transfer explicitly disabled. Fortunately, most of these sequences could be encapsulated in library routines, so pre-emptive multitasking was chosen in preference to a coroutine system.

5.3 MPFW Execution

Upon launching, MPFW initialises the operating system and toolbox managers. If the program is launched with one or more sub-applications (by selecting them from the Finder), these are installed in a menu; otherwise the user is presented with the Standard File dialog to select the first sub-application. Others may be installed later by opening their files through a menu command. An instance of an installed sub-application is invoked by selecting it from the appropriate menu.

The system process stacks are then allocated, the system tables initialised, and the patches installed in the operating system. This all takes place under the context of the original stack, which retains control of the main loop process throughout the execution of MPFW. Finally the interrupt-level scheduler is installed, and the system enters multitasking operation mode.

There are five system processes. The main loop process performs most of the general housekeeping of MPFW. An idle process is set at the lowest priority; it can never be blocked and continually attempts to wake up the main loop. The remaining system processes - the low-level communications protocol handlers and the remote application service agent - are described in Chapter 6.

The main loop process holds the highest priority and is scheduled to run at every vertical blanking interrupt (60Hz) and whenever the system is otherwise idle. On each pass it collects and analyses an event, acting upon it and/or distributing it to one of the sub-applications' private event queues as appropriate. It also performs the following actions at given intervals:

- calling *systemtask* () ($1/60$ s)
- scheduling the I/O processes if there are new input or output requests pending ($1/60$ s)
- waking processes which are sleeping on a timeout semaphore ($1/10$ s)
- checking communications timers ($1/10$ s)
- resetting the cursor according to its current location ($1/10$ s)
- reaping dead processes ($1/10$ s)

Because these occur in the context of the main loop, sub-applications are freed from most operating system support requirements; they need not interrupt long calculations to call systemtask() or otherwise attend to the expectations of the user interface.

5.4 System Data Structures

Communication between the system processes and the sub-applications occurs through shared data structures (Fig 5.1, Fig 6.4). Most of these structures are variants of a common queue element type, which allows a regular method of formally transferring the information from the scope of one process to another. Such transfers must be protected from interruption, as must all alterations to structures which can be accessed from separate processes.

Fig. 5.1 MPFW Local Event Communication

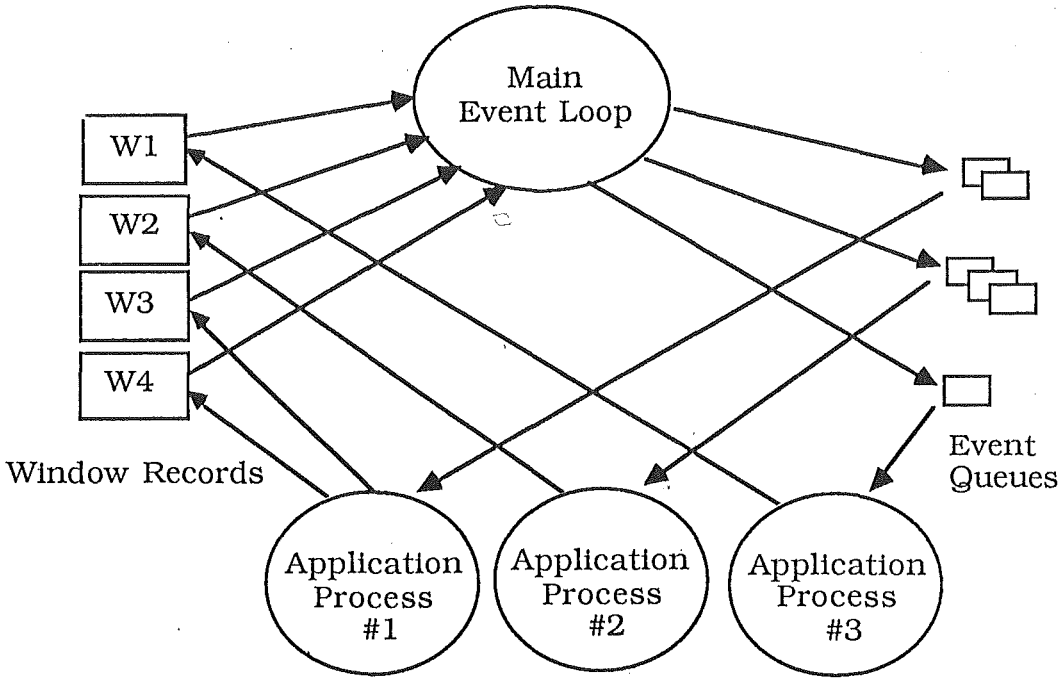
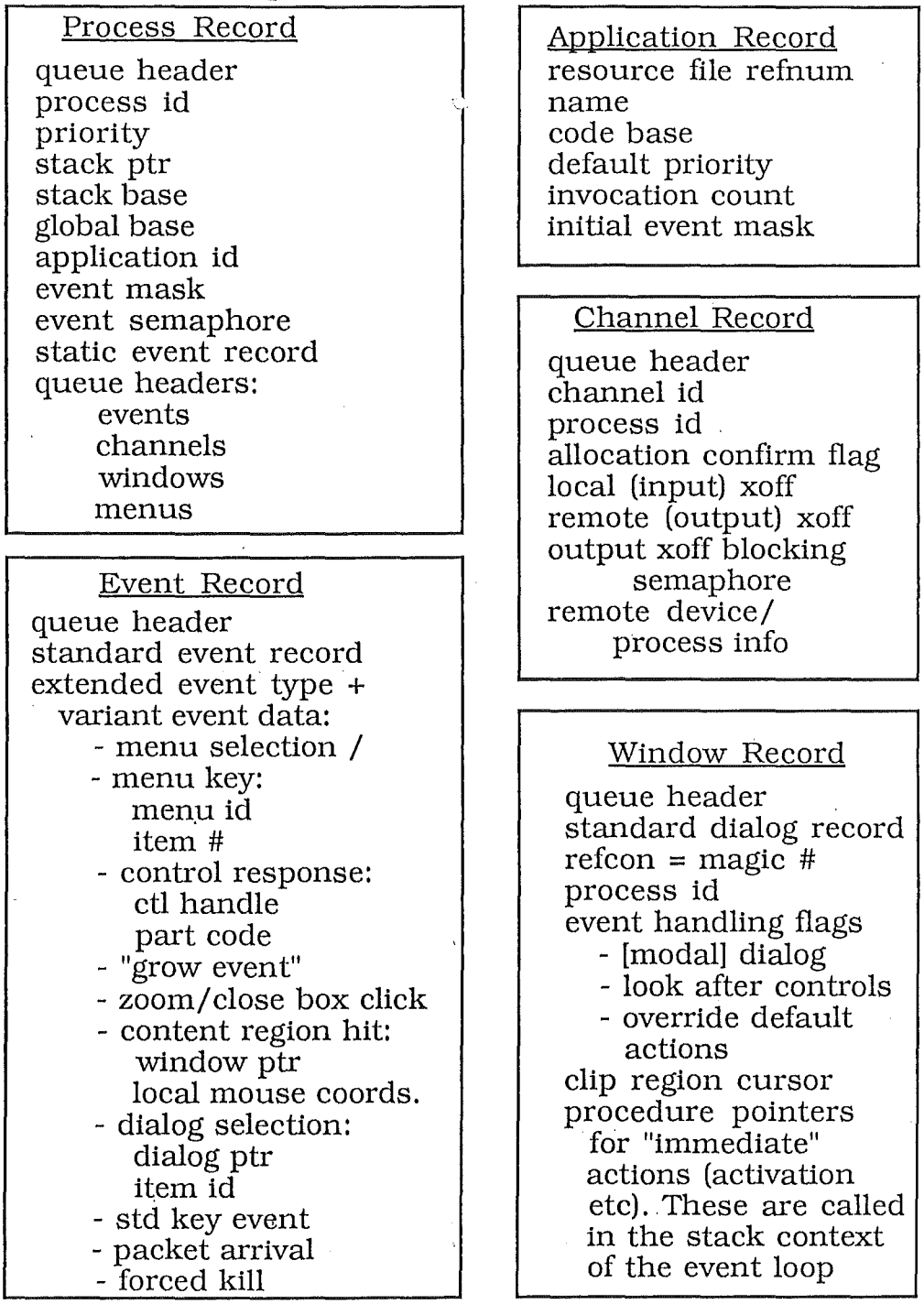


Fig. 5.2 MPFW Data Structures



The *Application Record* (Fig 5.2) is initialised from a resource held in the sub-application's file. It holds information used to initialise each invocation of the sub-application:

- sub-application name to be installed in the launch menu
- default priority, set by the programmer to reflect the nature of the program (eg higher value for interactive programs)

- maximum invocation count - some sub-applications will support only a given number of invocations from the same resource file (usually 'one or many'); the field allows MPFW to enforce this restriction
- initial event mask, indicating the classes of event which the sub-application wishes to receive
- resource file reference number (entered when the sub-application is installed)
- code segment base (entered at the first invocation)

The *application ID* is taken from the index of the pointer to the application record in an array called the*application table*. Similarly derived identifiers exist for processes (*pid* 's) and for channels. Channel records hold information required for the remote communications protocol described in Chapter 6.

Some of the data structures are extensions to standard Macintosh interface manager records. These extended window, event and menu records communicate information between the main loop and the sub-applications, as described in the following section.

The central structure is the process record. Part of this record contains information required by the process dispatch routine and the semaphore primitives:

- queue link for a run queue or semaphore
- stack base and saved stack pointer when not running
- global data segment location
- current dynamic priority
- sub-application identifier, for initialising the process and restoring the default priority

Other process context information - the registers, current port, resource file etc - is saved on the stack. The process record also associates other interface structures with a particular process: queue headers for communications channel structures, extended window records, owned menus and the sub-application's private events are stored in it. There are other fields for event management, including the process's event mask and a semaphore to synchronise the transfer of event records from the main loop to the sub-application.

5.5 Event Analysis[1]

Events in MPFW fall into three categories:

a) those which are independent of any sub-application

b) those which are associated with a sub-application, but can be serviced entirely within the context of the main loop

c) those which must be passed to the sub-application's event queue

In the first category are menu selections for MPFW's own menus, events for desk accessories, and events for which there are no interested applications (discarded). These are handled in the normal manner. All other events can be associated with a particular window or process :-

- mouse button events are associated with the window in which the cursor is located, or with the frontmost (active) window in the case of menu selections. These can be found by the *FindWindow()* and *FrontWindow()* calls respectively.
- null events and keyboard events are associated with the frontmost window
- update and activate events are associated with a given window *per se*
- disk events in MPFW are associated with the first process which is waiting on the disk insertion semaphore

Network and driver events are currently discarded.

The extended window structure contains a number of flags and procedure pointers for modifying the main loop's standard event responses. The standard modal dialog handler does not allow the use of the edit menu and would have the undesirable effect of disabling background processing if used, so one flag requests that the main loop treat the window as would *ModalDialog()*, without the side-effects. Other flags signify modeless dialogs, which use the *DialogSelect()* routine to handle certain events, and *palettes*, which respond immediately upon a mouse click without requiring an initial selection click.

The procedure pointers are provided to allow the sub-applications to request immediate window manipulation actions in the context of the main loop, rather than waiting for the event to be passed to the private queue. These must be used with the same care as all asynchronous routines; in fact their usual purpose is to notify the

[1]Note: an outline of MPFW's event analysis algorithm is presented in the flow diagrams of Appendix V.

sub-application of changes effected by the main loop's default actions. They would be of more use if implemented as UNIX-like signals able to modify the stack context of the target process, but this capability is not yet available in MPFW .

Activation events are of special importance to MPFW. When a new window is brought to the front, its owning process takes control of the menu bar and some menus may have to be altered or replaced. The extended window record includes flags which are used to reset the state (enabled or not) of items in the common **File** and **Edit** menus, and the private menus are restored from the list in the process record. Sub-applications gain a priority level when they own the active window, and this is adjusted during the activate and deactivate event responses.

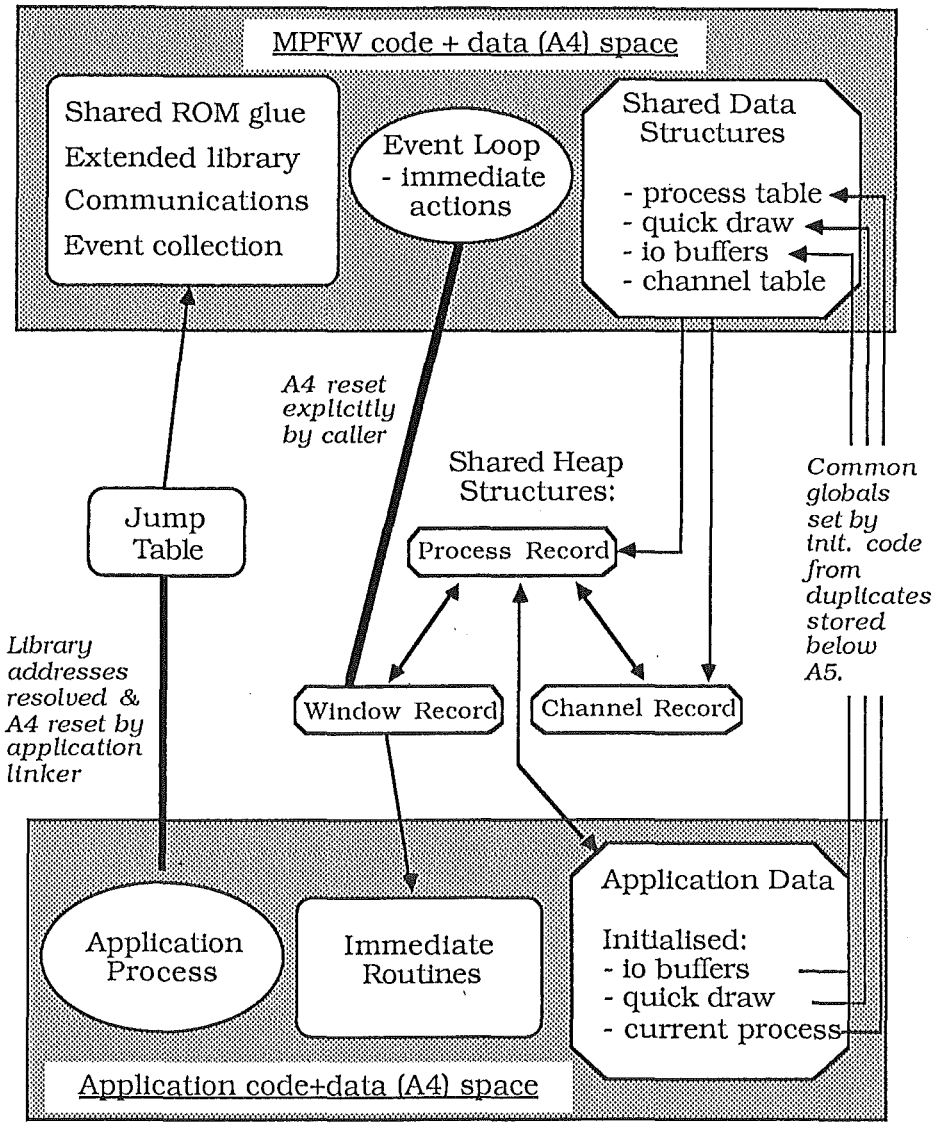
If an event cannot be handled completely by the main loop, it will be passed to the sub-application in an extended event record,which includes the information already gained about the event. The range of event types is also increased to allow for heavily pre-processed events such as menu selections, dialog item interactions and "go away" box hits. Three of the application defined events are used, but only in the sub-application queues. They are used to indicate "quit" commands, packet arrivals from the communications handler, and miscellaneous signals.

5.6 Data Space Integration

Because the sub-applications are separately compiled, the global data structures of MPFW are not directly visible to them. Although this information hiding is advantageous for modular decomposition, access to certain structures (e.g. the QuickDraw variables and the sub-application's process record) is necessary (Fig. 5.3).

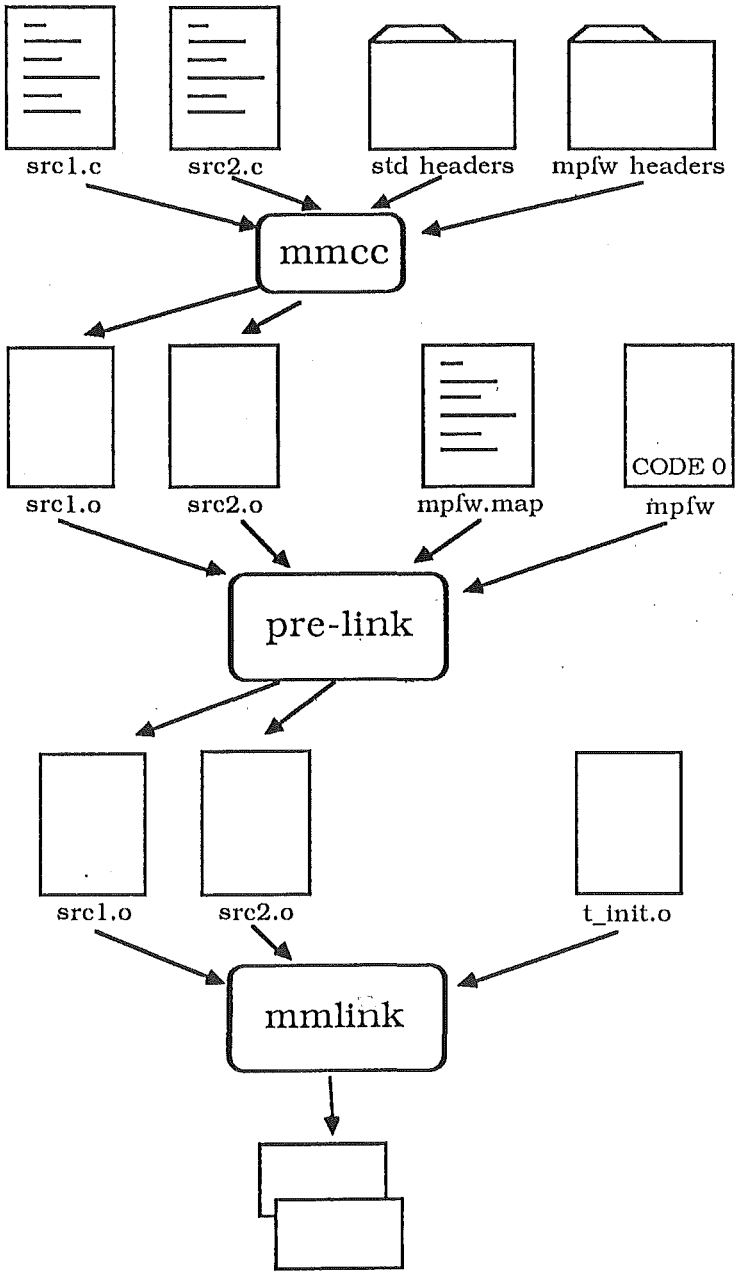
The sub-application linking process requires a special object file, *t_init.o* to replace the Megamax system library's application startup routines (Fig 5.4). This file loads and initialises the string and data segment which is referenced relative to A4. To enable access to shared structures, MPFW sets up the standard application global space (§3.2.2) to store their addresses. The sub-applications use modified header files with C pre-processor macros which redefine the structures to be indirectly referenced. The initialisation code can then simply copy the A5-relative globals into the appropriate locations in the private global space.

Fig 5.3 MPFW Global Data Space Integration



It is necessary to ensure that the appropriate A4-relative global space is in use when calling routines compiled with a different global scope. When the pre-linker resolves library calls in the sub-application, it also surrounds the calls with statements to change and restore the data segment base register. In the other direction, MPFW explicitly adjusts this register before calling sub-applications' action routines, using the given field in the process structure to accomplish this.

Fig 5.4 Creating an MPFW Sub-Application



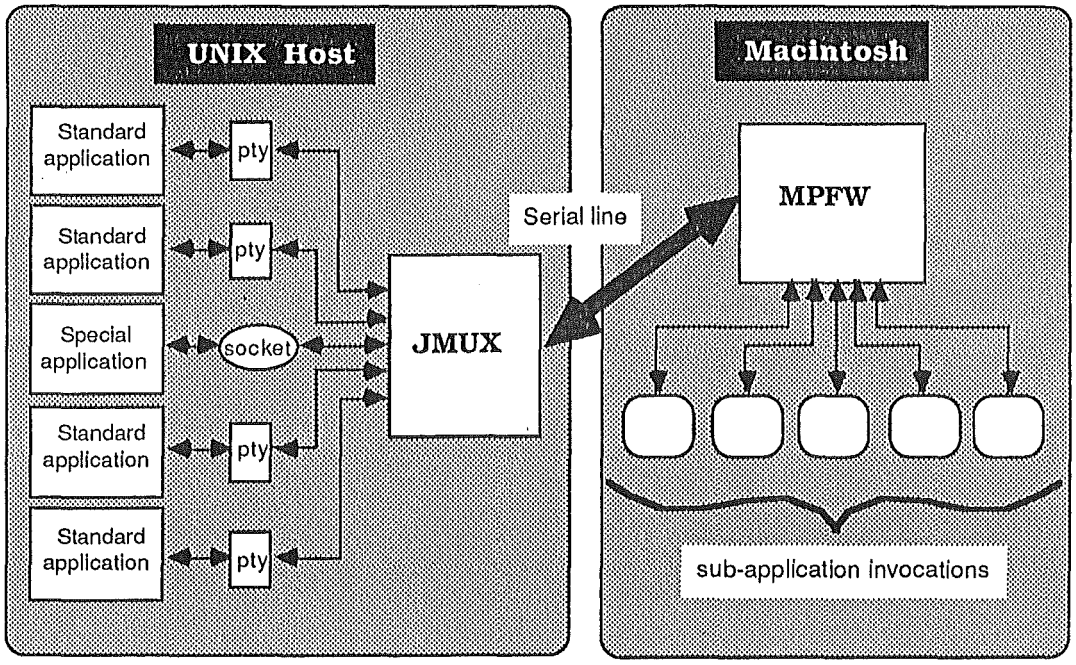
mpfw application

Chapter 6

Macintosh to UNIX Communications

Although MPFW offers many advantages for Macintosh programming in general, it was originally designed to assist the development of UNIX interfaces. This chapter describes the communications protocol used over the serial line between MPFW and the host multiplexor JMUX. The protocol provides up to 31 reliable data channels between MPFW sub-applications and host pseudo-terminals or UNIX domain sockets (Fig 6.1).

Fig 6.1 Communication between MPFW Sub-Applications and Host Processes



6.1 Protocol Selection

At the time when MPFW was designed, local conditions restricted the physical medium for communication to RS-232 serial lines (§1.1). Macscrewn[1] and UW use simple protocols based on unchecked escape sequences for most operations, but this was considered inadequate for MPFW because interface applications may require the exchange of large blocks of data which are not simply echoed to the screen (e.g. for file transfer or distributed editing packages). On a directly connected serial line

[1]Macscrewn uses XMODEM for file transfer.

transmission noise errors are rare enough to be ignored, but characters may be lost as a result of hardware overruns in the VAX dz-11 serial driver, which are quite common during block transfers at 9600 baud.

The requirements for the protocol were as follows:

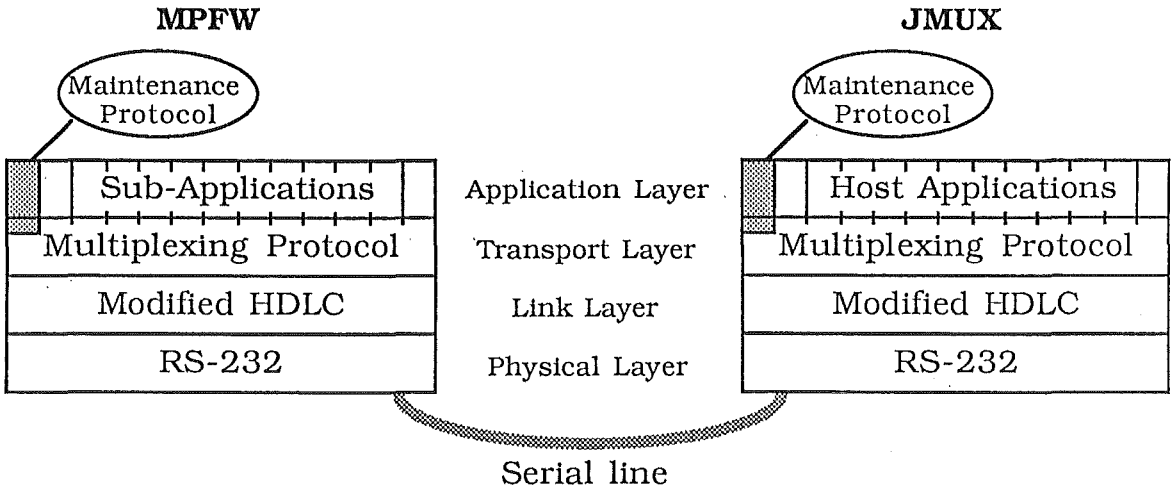
- reliable, sequenced end-to-end data transfer
- bit-stream or full octet data transparency
- error recovery, preferably by re-transmission since the predicted error pattern was one of occasional burst errors, rather than frequent single-character corruption
- multiplexing for at least 16 independent channels
- minimum throughput of 150 characters per second over a single channel when operating over a 9600 baud serial line
- maximum return delay of 0.3s for single character echoing
- independent flow control for each channel

The following attributes were also considered desirable:

- variable length block size - related to the efficiency requirements above
- out-of-band data transmission for interrupt characters etc.
- the ability to buffer blocks in the host driver, instead of waking up the host protocol agent for each incoming character. This objective was related to problems with the terminal servers on the Department's Cambridge Ring: *cooked mode* messages were buffered locally, but incoming characters in raw mode had to be passed immediately to the host. Raw mode block transfers caused a flood of small ring packets, and this often overloaded the servers which would then reset the channel and discard all input for seven seconds.
- separation of multiplexing functions from link-level services in order to allow subsequent installation of other communications systems (e.g. ethernet/IP or AppleTalk). The separation corresponds approximately to the distinction between layers 2 and 4 of the ISO/OSI reference model; the serial connection is intrinsically end-to-end, so networking services (layer 3) are irrelevant. This was not an overriding consideration in the choice of an existing protocol because (realistically) neither layer was expected to be replaced independently.

Few existing protocols appeared to meet the minimum specifications. For the data link layer HDLC [ISO 82] seemed the most suitable choice, since other protocols for reliable serial line communication (e.g. XMODEM) were heavily oriented towards file transfer, using large blocks with stop-and-wait acknowledgement[1]. Internet TCP and CCITT X.25 (level 3) offer channel multiplexing and related services (flow control, out-of-band data), but the use of either appeared immoderate given the limited requirements of the system at this level.

Fig 6.2 JMUX / MPFW Protocol Layers



Because no public domain sources of a suitable protocol were available, it was necessary to implement the communications software as part of the project. This was recognised as time consuming and peripheral to the thesis research, but it did allow development of a protocol which met the specifications without introducing unnecessary inefficiencies. The magnitude of the task was reduced by taking HDLC as the basis for the data link layer, although strict implementation of the standard was considered inappropriate since neither agent would be of use for general purpose communications. The remaining elements of the communications system were custom-designed (Fig 6.2)

[1]The Serial Line Internet Protocol (SLIP) might also be a suitable link layer agent, but it was not available during the protocol selection phase of the project.

6.2 Link Layer Protocol

The JMUX/MPFW protocol link layer includes a number of modifications to HDLC which enhance its efficiency and reliability under the given operating conditions. The limitations and capabilities of the serial drivers - particularly the UNIX (Berkeley) terminal driver - were a major influence on these modifications. [ref. Appendix I.5 for supporting technical documentation] This section does not describe the entire link layer protocol, but concentrates on the aspects which differ from HDLC.

The first problem was data transparency. It was predicted that bit-stuffing would result in unnecessary inefficiencies for a software implementation of the protocol where all other components of the system were oriented towards eight-bit data units. Octets would require splitting and re-assembly at each end of the link, and the receiver would have to test for framing sequences at each bit boundary. Another factor was that low level debugging would be difficult if octet boundaries did not reflect the underlying characters.

As explained in §6.1 above, there was a motive for adapting the protocol so that each frame could be buffered in the host terminal driver before being passed to JMUX. This can be achieved by using the end-of-frame character as the line separator with the driver set in *cooked* mode. Most of the other cooked mode i/o processing functions must be disabled.

HDLC flow control methods alone (RR/RNR) were found to be inadequate for the JMUX/MPFW link. For efficient file transfer, a maximum frame size of 128 characters is allowed, and the UNIX terminal buffer size is fixed at 256 characters - i.e. it can be filled by just two frames[1]. If flow control is available only at frame level, there is a significant danger that software overruns will cause data loss before the remote protocol agent can process the RNR frame[2]. Xon/xoff must be used for intra-frame flow control, because the dz-11 does not support hardware methods.

---

[1]This problem has also been noted in a recent USENET communication. The author was engaged in the development of a similar protocol, but resorted to reducing his frame size (with some performance loss) in order to preserve data transparency on the raw link. He called for user control over the terminal buffer size.

[2]The danger would be greatest in the presence of high data rates in both directions (as in the distributed editor of §2.5.5).

Occasionally, xon/xoff flow control breaks down because an xon signal is missed. In normal use, it is unwise for the system to attempt to recognise and correct this because the same mechanism is used for manual flow control (e.g. when the user is paging through a large document). This does not apply to MPFW, so xoff is cleared automatically after it has been asserted for an unreasonable period (1.5s). As with other timeouts, if this condition occurs repeatedly it is assumed that the remote protocol agent has failed, and the user is notified if possible.

This is not the end of the flow-control problems, however, because input flow control (*tandem mode*) in the UNIX terminal driver is not guaranteed to be reliable for our purposes. When the input character queue reaches the *high water mark*, an xoff character is sent if and only if there are characters which can be passed to the reading process (JMUX). Under the pathological condition where several corrupted frames (lacking terminators) are held in the input queue, no xoff is sent and a software overrun ensues. In raw mode, the driver simply flushes the excess characters; in cooked mode, a BEL character (#07) is transmitted for each extra character received. To clear this condition, the remote protocol agent (MPFW) must respond to the BEL by sending an interrupt character which invokes a signal handler in JMUX. JMUX then flushes the garbled input queue through a driver control call, and the link layer re-synchronises according to the normal methods.

Note in the above paragraphs that six characters have been reserved for control purposes:

- XON/XOF (xon and xoff for flow control)
- FRS/FRE (frame start and frame end markers - these must be different to avoid waking JMUX at the start of a frame)
- KOP ('kill output' - the character sent by the cooked driver when the input queue overflows - this is the only control character with a fixed value (#07))
- KIP ('kill input' - interrupt)

Some means of maintaining data transparency is therefore required. A seventh character is reserved as an escape (LNC - literal next character). For implementation efficiency, all special characters are chosen from a set of eight[1] octets with a common bit pattern. When one of these octets appears in the link layer data stream, it is preceded by LNC and its bit pattern is modified to remove it from the reserved

[1]The eighth reserved character is used to suspend the protocol agents for debugging purposes.

set. This process is reversed at the receiving end before the frame is interpreted. The bit modification is required as well as the escape because some of the special characters are recognised at driver level, and the Macintosh serial driver does not support built-in escape processing.

For efficiency, it would be desirable to choose the bit pattern so that reserved octets would be likely to appear infrequently (keeping in mind that the expected frequency of HDLC header bytes and ASCII printing characters is high), but the choice is constrained by the fixed value of the KOP character(#07). The following bit pattern is thus reserved:

0xxx 0111

where "x" denotes "don't care". When a data character is modified to distinguish it fron a special character, bit 3 is changed from 0 to 1. (N.B. LSB=bit 0; MSB=bit 7)

There was some difficulty in choosing an appropriate timeout for frame acknowledgement, taking into account variations in host performance under different loading conditions. If the timeout is too short, performance will degrade further through unnecessary retransmissions; if too long, the delay before a corrupted frame is retransmitted will be unsettling for the interactive user. The JMUX/MPFW protocol therefore uses a retransmission scheme which relies on timeouts as little as possible, using the optional REJECT frame. The mechanism is described below, as seen by the receiver of a stream of INFO frames:

In the absence of errors, each incoming frame is acknowledged with a supervisory frame (RR) or by piggybacking if possible. When an out-of-sequence frame arrives, a REJ frame is transmitted, bearing the sequence number of the next frame expected - N(R). The REJ causes the sender to perform the following actions:

a) if necessary, reset the sending window base to N(R) - as for any supervisory or info frame
b) flush pending output
c) reset the sending sequence counter V(S) to N(R). This causes retransmission of all pending frames starting at N(R)

The receiver ignores subsequent out-of-sequence frames until the expected frame arrives; if each buffered input frame resulted in a REJ, the sender would be forced to repeat the above actions unnecessarily. The REJ is repeated only in the case of a timeout, as described below.

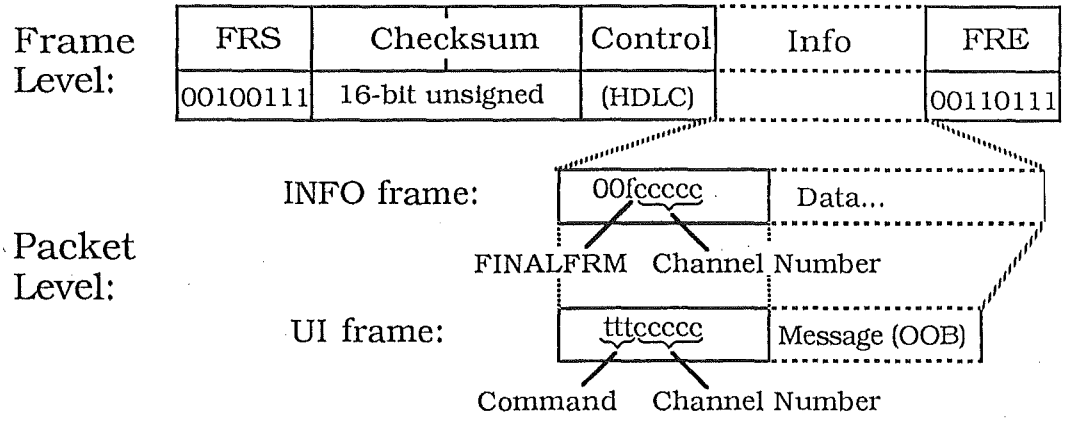
This mechanism is sufficient only if the missing or corrupted frame is followed immediately by another, and if the frame is successfully transmitted on the second attempt. Acknowledgement timeouts are still required. If the sender of an INFO frame has not received an acknowledgement within the allowed timeout, it sends a supervisory frame with the poll/final bit set. The receiver must respond to this with another supervisory frame, also with the poll/final bit set. If the value of N(R) on this frame fails to acknowledge the latest INFO frame, retransmission takes place as described for the REJ case above. If the receiver fails to respond after repeated polling (10 retries) the link is considered dead and the user is notified if possible. The type of supervisory frame used for the poll and response depends on the agent's current *rejection state* : if a REJ has been sent, but the expected frame has not been received, the REJ is repeated; otherwise an RR frame is used.

The above timeout behaviour is similar to that recommended in the HDLC standard [ISO 82 §6.2.4]; some naive implementations retransmit unacknowledged frames immediately without the exchange of supervisory frames. The more complex method is used in MPFW because a frame timeout will often signify temporary congestion in the underlying link (particularly in the Departmental LAN terminal servers) rather than frame loss or corruption. If congestion is the cause, flooding the link with retransmissions will be counter-productive. It is better to probe the link and ascertain the correct value of the remote receive counter before proceeding.

There are some deviations from standard HDLC in the frame format (Fig 6.3). The address field is not supported, and the frame check sequence appears before the control field rather than at the end of the frame, so that its offset from the start of the frame buffer is always the same. Also, the JMUX/MPFW FCS is a simple 16-bit unsigned checksum rather than the CCITT cyclic redundancy code specified in HDLC. This change was introduced for efficiency reasons, but it may be reconsidered if a higher level of reliability is required.

Unnumbered information (UI) frames are used to support single octet out-of-band data transmission, and are also used by the multiplexing layer for channel flow control. The out-of-band mechanism is effective, if a little heavy handed: all local buffered output and remote buffered input is flushed before the frame is sent; pending INFO frames must then be retransmitted.

Fig. 6.3 JMUX / MPFW Protocol Frame and Packet Formats



6.3 Multiplexing layer

This layer provides multiplexing and flow control for up to 31 full duplex data channels. Each channel is normally treated as a continuous data stream, although message boundaries are supported with packet fragmentation and recombination to take into account the limited size of an INFO frame. Because reliable end-to-end transfer and sequencing are guaranteed by the link protocol, these functions are not duplicated.

Each channel is identified by a number in the range 0-31. A channel cannot be used until it has been *opened*, which involves initialising a *channel record* in MPFW and JMUX, and ensuring that a process is attached to both ends of the channel (§6.4). Channel 0 is reserved for control information, and is opened automatically as soon as the link layer is ready.

All link level INFO frames hold a packet for the multiplexing layer. A packet starts with a single character header which contains the channel number in bits 0-4. Bit 5 (FINALFRM) is set if the packet contains a complete message or the final fragment of a message; the remaining bits are not examined but should be kept as zero to allow future protocol extensions. Messages may be fragmented and recombined before being passed to the upper layers, according to the following conventions:

- the data sent by a sub-application Send() call (§6.5) is regarded as a single message. If it will not fit into a single frame, it is fragmented as above with the FINALFRM bit set only on the last fragment. (N.B. The 128 octet frame limit includes headers and escape characters, so the available packet size may be as little as 60 bytes).
- fragments of a message are recombined in order to fill the buffer specified in a sub-application Recv() call (§6.5). Separate messages are never recombined, so Recv() always returns when a FINALFRM packet has been read.
- host pseudo-terminals are regarded as stream devices, so message boundaries are ignored. Packets are written to the device as soon as they arrive, and all packets sent out are treated as complete messages.
- processes which communicate with JMUX through UNIX domain sockets are required to use the Sun RPC Record Marking Standard [SUN 86b], in which all records are preceded by a header containing length and fragmentation information. RMS boundaries are maintained across the JMUX/MPFW link, so that only the final fragment of an RMS record has the FINALFRM bit set. Because an RMS record may be longer than the maximum size allowed for an IPC transaction, JMUX does not combine incoming fragments but generates a separate RMS header for each, thus leaving recombination to the receiving application.
- channel 0 (control) messages are always recombined before being passed to the appropriate handler.

As mentioned in §6.2, link level UI frames can be used to carry out-of-band data and flow control information for the multiplexing layer. The first octet in the UI data field contains the channel identification - as for an info frame - and the message type. The following messages are defined:

- STOPCHAN: halt output on the specified channel. (Equivalent to xoff.) STOPCHAN is sent out automatically when more than two events/packets are waiting to be transmitted to the sub-application/host process. In JMUX, xoff is raised in the receiving pseudo-terminal master, and no further read()'s are performed until the condition is cleared. In MPFW, the sub-application will block during any subsequent Send() calls on the given channel.

- STARTCHAN: restart output on the specified channel. (Equivalent to xon.) STARTCHAN is sent out automatically when the appropriate event/packet queue length has been reduced to one. STARTCHAN clears the effects of STOPCHAN.
- POLLCHAN: check input flow control status on the specified channel, and respond with STARTCHAN or STOPCHAN as appropriate. POLLCHAN is transmitted periodically on all channels which are waiting for a STARTCHAN. This avoids locking up a channel because of a missed STARTCHAN; reliable delivery is not guaranteed for UI frames.
- OOBCHAN: the second octet in the UI data field contains a character which is to be passed to the appropriate process immediately. Out of band data transmission is implemented only in the MPFW→JMUX direction, and is available to MPFW sub-applications by means of the SendOOB() shared library call .

6.4 Channel Allocation and Process Maintenance

This section describes how channels are allocated and released, and how connections between MPFW sub-applications and host processes are formed. Currently there are only two methods of forming such a connection.

An MPFW sub-application can request a channel connected to a pseudo-terminal master device. If a command string is provided, the specified command is executed with standard input, output and error attached to the pseudo-terminal slave. The request is made through the call

```
int             /* returns the id of the new channel, or (-1) on failure */
openchan(rdev, rcmd, chutmp)
int rdev;       /* host device type; DEV_PTY or [soon] DEV_USOCK */
char *rcmd;     /* command to be launched . (optional for pty) */
int chutmp;     /* (pty only) if true, modify the host's /etc/utmp file to divert
                   system messages etc to this channel */
```

The sub-application can determine the name of the remote device (pseudo-terminal slave name or socket address) and other information about the channel by examining the channel record (§6.5).

A host process can request a channel connected to a newly-launched sub-application. JMUX maintains a UNIX domain server socket, through which other processes can obtain a socket connection. The first record sent through the socket must contain the name of an MPFW sub-application which is to be launched with the

new channel already connected. Sub-applications which can be launched in this way should check for existing channels before calling openchan().

The protocol for allocating a channel and/or executing a remote command is conducted through the maintenance channel (channel 0). The procedure is as follows:

a) the caller (JMUX or MPFW) searches its *channel table* for a free channel. To minimise the risk of collisions, MPFW prefers low-numbered channels and JMUX prefers high-numbered channels[1]. A new channel record is allocated, and its address entered into the channel table.

b) the caller builds a *channel registration request* packet, and sends it on the maintenance channel. The request packet contains the selected channel number and any information which may be required to make the remote connection; this may include such things as the device type (pseudo-terminal, socket) and command or sub-application to be executed on the channel. Some information about the local connection (e.g. the connecting device name) may also be included. At this point, the openchan() call returns the number of the selected channel, but any attempted operations on the channel will block until remote confirmation is received. This is ugly, and should be changed so that openchan() blocks throughout the sequence.

c) the callee checks that the channel is free, and if so it installs the new channel record and performs the necessary actions to set up the remote connection. It then builds a channel registration response packet, and returns it on the maintenance channel. If a collision occurs, or the connection procedure fails (e.g. if the command is invalid), the response packet contains a message to this effect; otherwise the details of the connection (device name etc.) are returned, and the callee marks the channel as active.

d) if the connection is successful, the caller marks the channel as active; if not, it frees the channel record and informs the requesting agent of the failure: in MPFW, a SIGCHAN signal event (q.v.) is posted; in JMUX, the socket connection is terminated.

[1] After the method used in X.25

When JMUX or MPFW closes a channel, it posts a *channel release* packet on the maintenance channel. No response is required. Channels are closed under the following circumstances:

- when an MPFW process dies or explicitly closes a channel. If the channel is connected to a socket, the connection is terminated; if to a pseudo-terminal, JMUX writes an end-of-file character and then closes the master device.
- when a host process launched on a pseudo-terminal dies, or when a socket is closed by the connecting process. In either case, MPFW will post a SIGCHAN *signal event* (§5.5) to inform the peer sub-application that the channel is no longer available; usually the sub-application will respond by exiting.

The range of connection mechanisms should be extended. It would be a simple matter to allow the launching of a host process on a *socketpair* rather than a pseudo-terminal, and this should be done to allow direct invocation of socket-based host applications from their MPFW counterparts. Ideally, it should be possible to form a connection between an executing MPFW sub-application and a host process, using the semantics of the Berkeley IPC server/client model [UCB 86b]. This would require some extensions to the protocol and the MPFW communications library.

Seen in retrospect, this "maintenance layer" is an unfortunate deviation from recommended practice in that it provides both application-level and transport-level services. In the ISO/OSI reference model, each layer should be responsible for opening and closing its end-to-end connections. The JMUX/MPFW protocol breaks this rule in order to simplify the launching of an application on a new channel, but by doing so it makes the implementation of other connection mechanisms more difficult.

6.5 MPFW Communications Implementation

Three MPFW system processes are dedicated to the support of the JMUX/MPFW communications protocol, although some protocol operations are performed in the context of other processes. Separate processes are warranted for independent continuous operations, particularly those which must maintain large amounts of contextual information while waiting on external events. The multiplexing layer does not meet these criteria, so its functions are shared among several processes; the only contextual information required is for message splitting and recombination, and

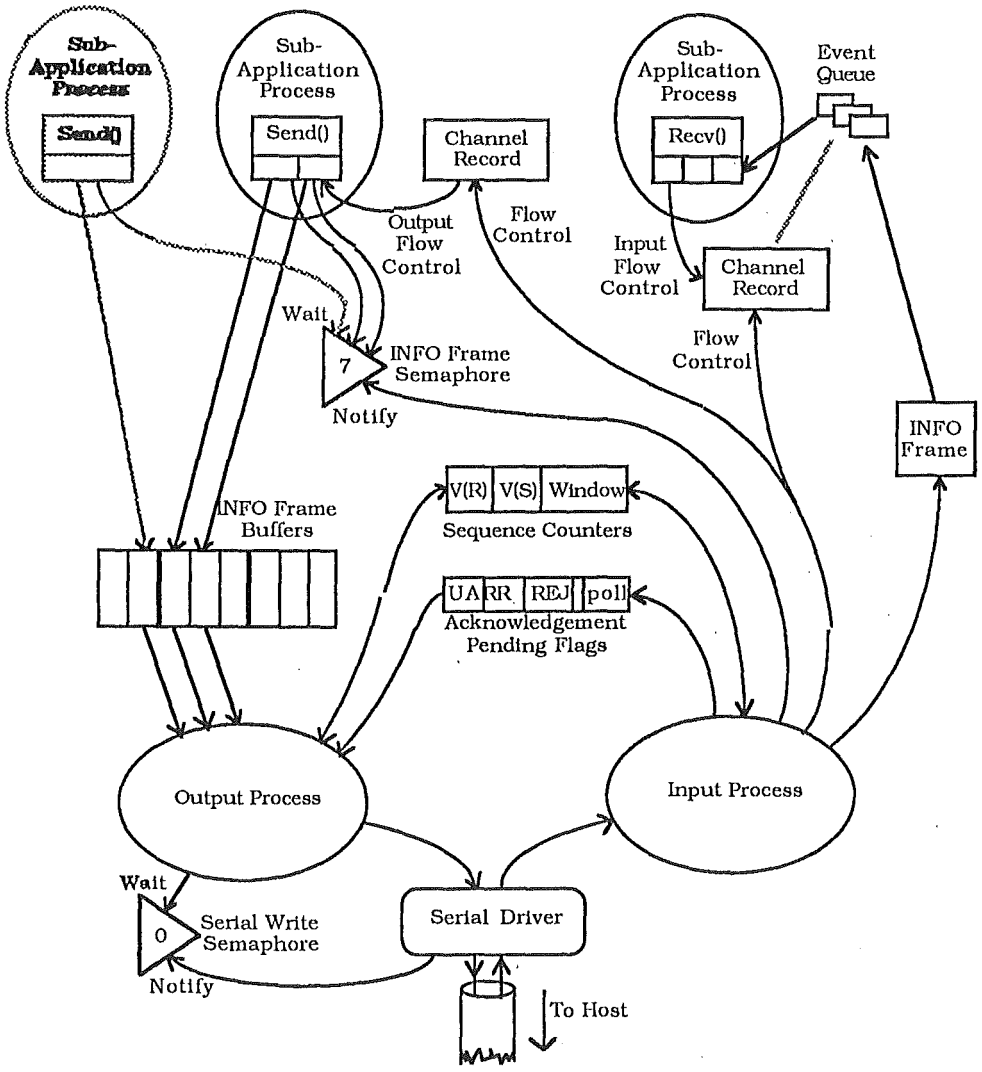
this can be held in the calling process. The division of process responsibilities is also influenced by synchronisation issues - for example, link-level input processing must continue while output is blocked to avoid protocol deadlocks.

At sub-application level, host communications are performed through the shared library calls Send(channel, buffer, bufflen), and Recv(channel, buffer, bufflen). Any splitting or recombination of packets is performed within these calls - i.e in the stack context of the sub-application process. Recv() accumulates packets and writes them to the given buffer, blocking on the calling process's event semaphore until its return criteria are met. Send() divides the given message into packets, passing them to the link layer through shared INFO frame buffers. Eight frame buffers are provided - one for each link-level sequence number. Send() writes directly into the packet field of these, inserting escapes and generating a partial checksum. Before gaining access to an INFO frame, Send() must wait on a buffer allocation semaphore. This semaphore has an initial count of seven and is notified by the input process whenever a new acknowledgement arrives, thus ensuring that each buffer's contents are protected until the frame is acknowledged. The sequence number of the next free buffer is maintained by Send(), and a "frame-in-use" flag is set when the frame is ready for the link layer.

In the communications sub-system, a number of flags and buffers are shared among two or more processes (as in the preceding paragraph). Operations on such variables are always protected, either by disabling process exchange or by providing a dedicated semaphore if long-term exclusive access is required. Specific access control mechanisms are not described explicitly in this section.

The *input process* reads characters from the serial driver, stripping out escapes and dividing the input stream into frames. It performs the link-layer management functions which are required in response to each valid frame (updating counters, signalling that acknowledgements are required etc.), and all multiplexing functions apart from packet recombination. The two layers are divided at procedure level.

Fig. 6.4 MPFW Communications Processes - Link and Multiplexing Level



To avoid re-copying data, the input process pre-views each frame's control field and directs INFO frames to dynamically allocated buffers. As these buffers pass up through the system, the (buffer address, buffer length) pairs are modified so that each layer sees only the relevant section of the frame. After the input process has identified the channel number for a given packet, it posts a *packet arrival event* in the event queue of the receiving process. The event record contains the channel number, the FINALFRM flag, the length of the data, and a pointer to the buffer.

The *output process* is responsible for writing all frames to the serial driver. Sequence counter values and requests to send supervisory frames are communicated from the input process to the output process through a set of shared flags (Fig. 6.4). Unnumbered frame requests are passed by a similar mechanism. INFO frames within

the sending window are taken from the frame buffer array and transmitted in sequence from V(S) to the most recently filled frame; the input process signals re-transmission requests by aborting the current frame and resetting V(S) (§6.2). The value of N(R) and the final checksum are inserted in the frame immediately before it is sent.

In order to allow other processes to continue while output is pending, the output process performs its serial driver write() calls asynchronously. Multiple calls are not queued, however, because the value of N(R) should be set as near to the time of transmission as possible, and because high-priority frames (unnumbered frames, poll responses etc) should not be delayed unnecessarily. The output process therefore sleeps on the *line semaphore* during each write, and is woken by the call's completion routine when the operation finishes.

Protocol timeout actions are initiated in the main event loop process. Timer variables are set by the input and output processes, and are checked every $^1/_{10}$ s(§5.3). The main loop is also responsible for waking the appropriate processes when new input or output is detected.

The *maintenance process* interprets and acts upon all incoming messages from Channel 0. Unlike the other system processes, its structure is similar to that of a sub-application in that it is driven through its event queue, and does not require special scheduling. It handles all the remote maintenance functions of §6.4, except for the initial transmission of requests to open or close a channel.

6.6 JMUX Implementation

The JMUX/MPFW protocol underwent substantial changes during its development. Three early communications systems were abandoned, some because the protocol was considered inadequate, others because of complexity. In these versions, the host agent was coded in a conventional manner without multi-threading.

Once the protocol design had been finalised, it became apparent that the standard "hierarchy and sequence" structure of a procedural program was painfully inadequate for a communications system. When compared to the methods available for MPFW, efforts to map a conceptual model of the protocol to the host agent's implementation were restricted. The maintenance of state information in several

continuous operations is an even greater problem in JMUX because the application-level entities are UNIX utilities running as separate system processes; it is therefore impossible to perform multiplexing layer functions in the application context. It was necessary to make repeated transfers of control among the various tasks, and to maintain state information for each layer and channel in static structures. In simpler terms, the program was a mess.

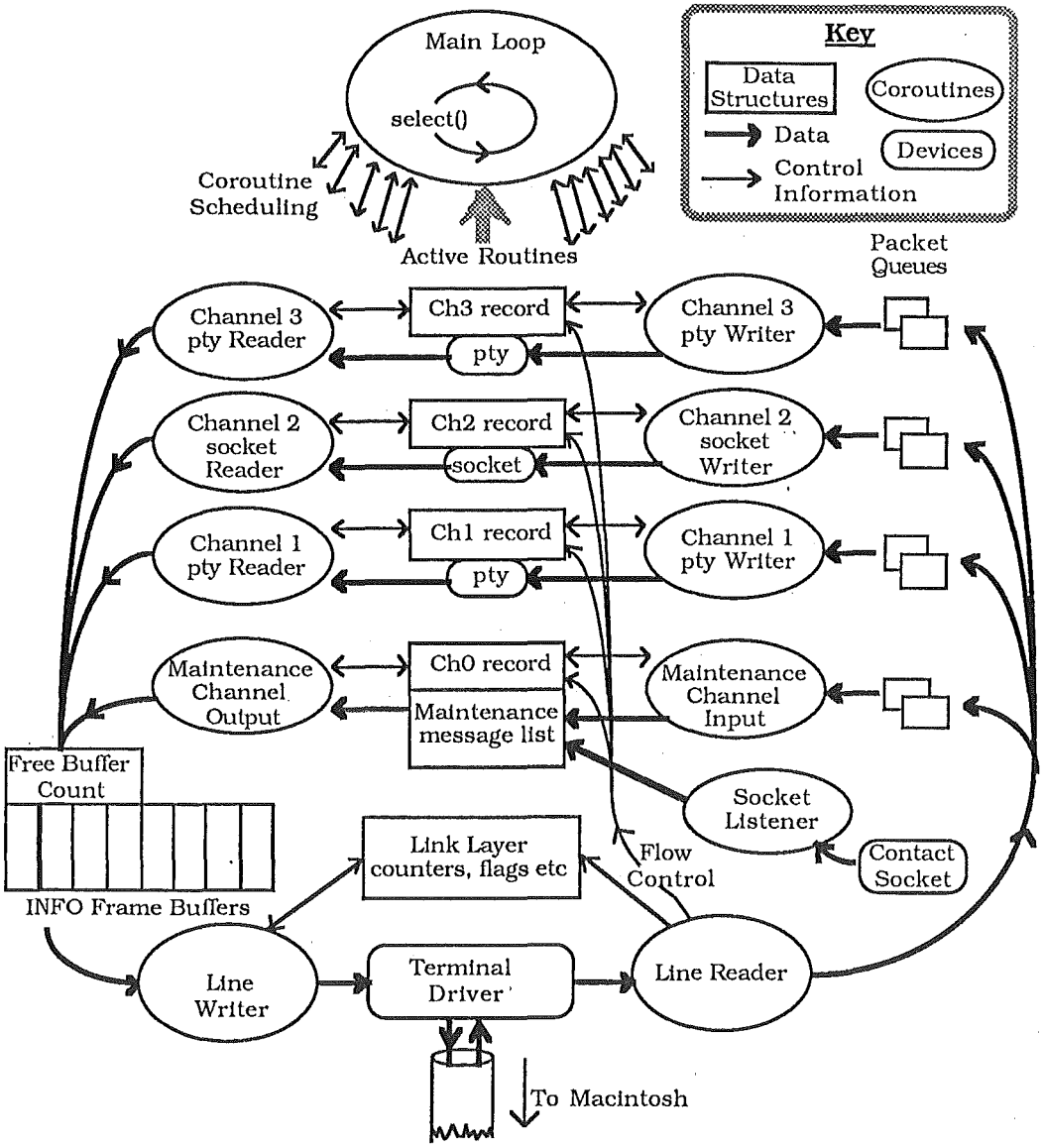
A common solution to this problem in UNIX communications applications is to divide input and output functions into separate processes (e.g. RLOGIN). This option was unsuitable for JMUX mainly because of the close interaction between input and output in HDLC - the two handlers really need a common data space.

What was required was a coroutine package for the host. Pre-emptively scheduled threads were considered neither necessary nor desirable for JMUX. Their advantages in MPFW were more relevant for sub-applications; in a pure communications system, the problems of shared data protection prevail and voluntary task exchange (using sleep() or resume() primitives) is preferable.

Unfortunately, C coroutine packages are inherently non-portable. Various people (including some recent USENET contributors) have proposed ingenious coroutine schemes which use the standard setjmp() and longjmp() calls as primitives, but these invariably fail if longjmp() unwinds its stack destructively - as it does on the VAX. A certain amount of machine dependent programming is required when performing the context switch, and when initialising the coroutine stacks.

The author has contributed to submissions in the USENET forum calling for a standard C coroutine library, but for JMUX an immediate solution was needed. A coroutine package was written, together with the necessary development support for in-line assembly inclusion. There is a certain amount of disagreement on the subject of ideal coroutine semantics but this is not relevant to the thesis, save to say that the coroutine package provides primitives for transferring control explicitly to another coroutine (transfer()), or for relinquishing control to the last explicit caller (sleep()). A single integer message can be passed at each exchange.

Fig. 6.5 JMUX Coroutine Structure



In most aspects, the structure of JMUX is very similar to that of MPFW's communications sub-system. Extra coroutines are required to handle input and output to the client processes' communicating devices (pseudo-terminals and sockets), and the maintenance channel handler is divided for regularity with this scheme (Fig 6.5). Another coroutine is required for the contact socket - it runs the standard non-blocking accept() loop of a Berkeley IPC server [UCB 86b], and initiates local channel allocation requests.

The main difference is in the scheduling of the coroutines. A central loop makes repeated calls to *select()* [1], checking -

[1] [ref. UNIX programmers' manual, section 2].

- which open descriptors - line, pseudo-terminals, and sockets - have input pending
- whether write operations are possible on those descriptors whose handling routines have requested output clearance
- whether exceptional conditions - I/O errors, socket terminations etc. - are present on any descriptors

The parameters of the select() call are set so that the call blocks until an I/O operation is possible. This avoids keeping JMUX in an expensive loop when it would otherwise be idle. Select also provides a timeout option, which is used to break the call if a timeout recovery action becomes necessary. An exception to the blocking rule is made if any coroutine is engaged in an activity which does not require output clearance or input before it can resume.

After select() returns, the main loop checks for timeout conditions and then works through the returned descriptor sets, transferring in turn to each coroutine which can continue. The coroutines thus called perform a single action, such as posting a frame or making a write() call, before returning control to the main loop. Effectively, the system does round-robin scheduling of all active coroutines, and while this may be primitive it has proved adequate in tests to date.

6.7 Protocol Evaluation

The JMUX/MPFW protocol has been implemented, and although it has not been tested exhaustively, it appears stable enough for practical use. The procedures used in debugging actually put greater strain on the system than would be expected in normal use - JMUX was running under the control of a debugger sending diagnostics to a remote machine, and the terminal line i/o was filtered through a pseudo-terminal to allow monitoring with a software protocol analyser. Both ends were repeatedly suspended to allow check-point debugging, and the timeout recovery mechanisms coped with this abuse.

In tests of a simple terminal emulator, the performance meets the given specifications (§6.1) with ease - exceeding the author's original expectations. Scrolling the contents of a host text file suggests an effective data rate of 250-350 characters per second, although many factors influence this figure and some flow-control packets are observed. Round-trip delay is just noticeable, but line input will be edited locally in most sub-applications and so the inefficiency of sending single

character packets while typing will not be applicable. There is room for improvement in the host scheduling - currently there is no way for JMUX to give priority to the active window's device, so two windows receiving continuous data streams scroll at the same (reduced) rate. This could mar interactive performance during file transfers.

There is, however, one major disadvantage to the protocol. Many institutions (including the Department of Computer Science) now have Macintoshes connected to their UNIX systems through local area networks which are capable of much higher data rates than serial lines. To allow MPFW to take advantage of this, it will be necessary to install a protocol based on TCP/IP[1]. If an implementation of the Serial Line Internet Protocol is available at that time, it may be prudent to replace the serial communications mechanism so that upper layers can share the same interface.

[1]Library support for TCP/IP on the Macintosh is under development in the US.

Chapter 7

PTYD - A Pseudo Terminal Server for BSD UNIX

7.1 Motivation

Pseudo-terminals are BSD UNIX device pairs which provide inter-process communication while simulating the system call interface of the terminal driver. Many interactive UNIX programs require the ability to manipulate terminal driver parameters, and will not work if their input and output are filtered through normal IPC mechanisms such as pipes and sockets. The slave half of a pseudo-terminal appears identical to a terminal device, but it is attached to a paired master device instead of a serial line driver. That is, the slave's input appears as the master's output and *vice versa.*

Remote login servers and window multiplexors such as UW and JMUX use pseudo-terminals to provide the interface for interactive sessions, and in this context they are indispensible. However, the procedure for obtaining a pseudo-terminal is complex and clumsy. The master device is permanently set in exclusive-access mode, making it impossible to open if it is already in use. Attempts to open the master device will also block if the corresponding slave device has not been opened - it is unclear whether this behaviour is intentional. The server application must therefore work through the list of pseudo-terminals, attempting to open each slave and master in turn until a free pair is found.

Another more serious problem occurs in the case of serial-line window multiplexors, where the user must be logged in through the serial line before the multiplexor is started. UNIX maintains a file of terminal sessions, /etc/utmp, which maps each active terminal to its current user. The entries are formed by the LOGIN program, and are removed by INIT when the user logs out (ref. Appendix 1). Communications programs such as WRITE, TALK and WALL (for system administration messages) use this file to locate users' terminal devices. If a window multiplexor is active, output from these programs will be directed to the original line which is running the multiplexing protocol. The results of this are undesirable, so most window servers disable write access to the serial line and prevent use of the communications programs. Attempts to redirect the programs by specifying the appropriate pseudo-terminal fail, and in the case of TALK it is also impossible to

initiate a conversation from a pseudo-terminal without a utmp entry; the program responds with the diagnostic message "You don't exist. Go away."

One solution to this is to allow the window server to modify the utmp file, removing the original entry and replacing it with a nominated pseudo-terminal. If such privileges are to be granted anyway, it would also be desirable to change the ownership of the pseudo-terminals and thus give the user control of their access modes. This also allows extension of the standard terminal security mechanisms (i.e. denial of general read/write access) to pseudo-terminals.

Setuid root programs are always a security risk, however, and system managers may well be reluctant to install third party programs of this kind when they must take responsibility for examining the source for possible loopholes. Window multiplexors are complex programs which must fork and execute user commands; this makes them even more dangerous and difficult to verify. Under this scheme they would also have be exceptionally robust in order to roll back the ownership and utmp changes before termination. To be strictly secure, they would also require operator intervention when left in an unresponsive condition (e.g. when accidentally invoked without the Macintosh agent), because they could not drop effective privileges to catch TERMINATE signals without also being vulnerable to user-generated KILL signals.

7.2 Package Description

PTYD is a root-privileged server daemon which provides a solution to these problems. Instead of searching for a pseudo-terminal and then performing the ownership and utmp changes internally, the client application (window multiplexor etc.) passes a message to the server, requesting exclusive access to a pseudo-terminal and the associated system changes. In UNIX domain IPC, file descriptors can be passed between two unrelated processes through sockets, so the server is able to return the master-side descriptor directly to the client process.

When the client has finished using the pseudo-terminal, it sends a message informing the server that the terminal ownership and the original utmp entry can be restored. A third message performs this action for all pseudo-terminals passed to the client. It implies that the client requires no further services and that the server can dispose of its internal session record, so this message should always be sent before

the client exits. Failure to do this will only result in a temporary inconsistency, however, since the server periodically checks the status of its clients and cleans up after any that have died.

The messages are encapsulated in three library routines which are available for client programs. The socket communications are handled internally by the routines, so no extra support is necessary. The routines are:

```
int                     /* returns 0 on success, -1 on failure (code in errno) */
getpty(masterfd, slavename, reset_utmp)
int *masterfd;          /* returned master file descriptor                     */
char *slavename;        /* name of slave ie "/dev/tty[pqrs][0-9a-f]"           */
int reset_utmp;         /* flag specifying whether to replace the              */
                        /* session terminal record in /etc/utmp with           */
                        /* the allocated terminal; reverts when                */
                        /* the pty is released                                 */
```

Getpty() returns (through reference parameters) a pseudo-terminal master-side descriptor and the name of its associated slave. The slave device owner and mode are changed as follows:

- the owner is changed to the effective owner of the calling process
- in 4.2 BSD, the group remains unchanged and the mode becomes 0622 (rw--w--w-)
- in 4.3 BSD, the group is changed to "tty" and the mode becomes 0620 (rw--w----)

If "reset_utmp" is specified, the pseudo-terminal is recorded as the session terminal in /etc/utmp.

```
closepty(masterfd)
int masterfd;           /* master fd of pty tobe closed and restored */
```

Closepty() closes the master descriptor and informs the server that the client is no longer using the terminal. Terminal ownership is revoked, and utmp restored if necessary.

```
ptyexit()
```

Ptyexit() does the equivalent of closepty() for every pseudo-terminal passed to the client, using a single server message. It also removes the calling process from the server's active client list.

7.3 Security

Although PTYD is relatively safe because it does not execute other programs or write files other than /etc/utmp, some extra measures are included to prevent fraudulent use of its capabilities. The author has notified appropriate authorities that efforts to improve terminal security in 4.3BSD have not been wholly successful, but it was seen as undesirable for PTYD to provide an alternative loophole in this area. While a detailed description would be inappropriate here, the problems are obscure and most unlikely to be inferred from a summary of PTYD's protection mechanisms.

Client processes must go through a "registration" protocol before any services are offered. This protocol ensures:

a. that the client has specified its true user id.

b. that the process id specified in the registration record has the same owner as the client. It would be preferable to prove that this process id really refers to the client, but this is very difficult, and in any case the privileges extended to the client could be passed to another process owned by the same user.

After registration, the client is given a unique token (a file descriptor) which is used as proof of identity in future transactions.

Another security enhancement is that PTYD makes a vhangup() system call before delivering a pseudo-terminal. This should deny continued terminal access to existing background processes which might otherwise interfere with the new session.

7.4 Acknowledgement

The code in PTYD for locating the pseudo-terminals and updating /etc/utmp was adapted from version 3.1 of J.D. Bruner's program UW (a more recent version than [BRUNER 86]). The acknowledgement and copyright conditions are carried in the source.

Chapter 8

Conclusion - Directions for UNIX and the Macintosh

In the course of this thesis I have outlined some of the difficulties inherent in attempting to give UNIX a user interface which could approach the ease of use and mass appeal of the Apple Macintosh. I have also criticised what I consider to be poor design decisions in the Macintosh's operating system and interface programming facilities.

MPFW addresses both of these issues. It offers the programmer a mechanism for applying specific features of the Macintosh user interface to individual UNIX applications - a level of integration which has not been seen, to my knowledge, in other Macintosh/UNIX packages. As a Macintosh programming framework, MPFW also alleviates many of the difficulties of Macintosh application development.

Multitasking has been one of the major themes of the thesis. This may seem surprising, given that the techniques are hardly new, but they are of current interest in the context of user interface management systems. Secondly, at the start of the project multitasking was neglected in most microcomputer operating systems, partly because of the memory limitations of the microcomputers which had existed until that time. Today there is still considerable disagreement as to whether pre-emptive multitasking is desirable for single-user microcomputers such as the Macintosh. As stated throughout the thesis, I regard it as highly desirable.

Shortly before this thesis is submitted Apple will release AUX, a version of UNIX for the Macintosh II (MC68020 CPU). According to USENET reports from Apple's beta-release testers, AUX will support the Macintosh interface toolbox as a set of library routines, and may offer a virtual machine environment to allow the running of existing Macintosh applications. If the virtual machine is successful, AUX will be of great benefit to the developer. But at this point, with the interface software and the UNIX operating system on the same machine, it may be necessary to review how the components of an interactive application should be distributed.

The facilities which make UNIX an excellent programming environment may be fundamentally unsuitable for general users; the existing utilities assume a level of skill which is inappropriate for casual computer users, and by presenting a high

level user interface for the utilities we lose the flexibility which makes them desirable. One view is that UNIX should offer the programming support and architectural framework under which dedicated interactive applications can be produced. Existing UNIX utilities can remain for the non-interactive mailers, communications systems and other background processes, but new user applications will be required.

Of all the attempts to create a suitable user interface management model, NEWS appears the most elegant and the most likely to gain the approval of the academic UNIX community. It also has the advantage of device independence through its use of POSTSCRIPT, which is achieving wide acceptance as a standard for graphical storage and communications. It seems, however, that UNIX user application structure needs redefining in terms of modern interaction facilities; mouse and bit-mapped terminal interfaces cannot easily be integrated into the original model of interchangeable I/O streams. With currently available systems, the most effective applications usually abandon the principle of dividing the core utility and the interface; as in Macintosh applications, the interaction support is embedded in the structure. I do not believe that this is strictly necessary, however.

As an example, it could be advantageous to consider separating the components of a word processor:

- one module for the device specific interaction components
- a second to maintain the internal document format, applying editing operations generated by the interaction module from a pre-defined set
- a third for the final document rendering on a given device.

The second module could then accept commands from another source (such as a document format translation program), or work with a range of front and back ends for different devices. It may be that the three components (particularly the interaction and maintenance modules) must interact at a level which makes separation through UNIX processes cumbersome (c.f. §6.6 - JMUX).

Perhaps the basic user-level "building block" should be moving away from the process and toward smaller units, such as sets of shared library modules with a common calling interface. Alternatively it may be worthwhile to preserve the UNIX process model but extend it through more sophisticated inter-process communications so as to support sets of cooperating processes working with

structured communications protocols. Shared memory designs and the associated synchronisation issues should also be explored, and the study of appropriate scheduling mechanisms for single-user workstations is important.

These are topics for future research. For the present, there remain large numbers of BSD UNIX machines and 68000-based Macintoshes. It is my hope that MPFW will prove successful as a means of enhancing the advantages of UNIX through the interface of the Apple Macintosh.

Appendix I

UNIX Operating System Architecture

This appendix is a brief introduction to the (Berkeley) UNIX concepts which are discussed in the body of the thesis. It is in no way comprehensive, and the reader is directed to the general UNIX literature in the references, particularly the two special editions of the AT&T Bell Laboratories Journal [BELL 78] [PIKE 84] and the 4.3BSD Supplementary Documents.

I.1 Processes

An understanding of the rôle of the process is central to an understanding of UNIX. A user command normally invokes a *process group* - one or more processes cooperating to perform a single operation. The relationship between these processes may be specified explicitly by the user, when several programs are combined in a complex command, or a given program may spawn several sub-processes to perform particular tasks. Typically, a user will run several process groups simultaneously:

- one group has control of the terminal device and is termed the *foreground* group.
- some perform non-interactive tasks in the *background.*
- some groups may have been interrupted from the keyboard and will be suspended awaiting a continuation signal. (BSD derived systems only)
- usually one process group will be blocked until the foreground operation terminates; this process group contains the command interpreter or *shell.*

Each process is assigned its own protected *user* address space for code, data and stack areas. Processes gain access to system resources through a clearly defined set of system calls. These are usually implemented as software traps, and when invoked they give the process access to the common *kernel* address space which contains the operating system code and data structures. The kernel maintains two major structures for each process: the *proc* structure which is locked in memory and only accessible to the process through system calls, and the *u* structure which is swappable and mapped read-only into the user address space. When a process is executing in kernel mode, it uses a stack in the *u* area.

A strict process hierarchy is maintained through links in the *proc* structures. Processes are spawned through the *fork* () system call, which creates a duplicate copy (child) of the current process, returning the process id of the child to the parent, and returning zero to the child in order to differentiate the two. After a fork(), the parent may either continue processing asynchronously or sleep in a *wait()* [1] system call until the child exits. In the former case, the parent is notified through a signal when the child exits, and is expected to collect the child's status information from a call to wait(). BSD systems allow processes to be suspended via signals; the parent may elect to be notified using the signal/wait() mechanism above when this happens,

Compiled programs are invoked through a system call (*exec* ()) which loads the requested executable file in place of the image of the current process. Most of the kernel attributes of the process remain unchanged. The exec() call also includes a mechanism for launching an interpreter on a given command file, thus allowing scripts to be invoked in the same manner as compiled programs.

At the root of the process hierarchy is the process *init*, which runs a *getty* process on each terminal line. When a user starts to log in, *getty* executes *login*, which confirms the user's password and sets its privileges to those of the user before executing the user's preferred shell. Commands invoked during the session are descendants of this shell. Any surviving children of a terminating process are adopted by *init* .

I.2 Signals

Signals are used to inform processes of exceptional conditions. Originally, there were few signals and they usually resulted in termination of the recipient process. Now there are over 30 defined signals in many systems, and they are used for such purposes as job control, I/O synchronisation, timing and notification of window size changes. In these cases, *handlers* are usually needed to perform the appropriate responses; these routines can be installed in various ways, and commonly alter the execution flow of the main program directly. BSD systems now provide atomic signal blocking operations to protect applications from race

[1] Some features of BSD's process control require an extended version of wait(), called wait3().

conditions. Signals can also be generated explicitly by user processes, and this encourages their use for process communication and synchronisation.

The extensions to the signalling mechanism have been criticised on the grounds that they have grown unsystematically and that they expose the application programmer to unnecessarily complex techniques. Current expectations of the user interface nevertheless demand some means of responding to external events, and signals provide the required functionality. However, alternative schemes may become necessary as interaction standards become more sophisticated.

I.3 File System

The concept of a file is extended in UNIX so that many OS entities are given the same system call interface and are identified through a the same hierarchical directory name space. Devices such as raw discs, terminals and tape drives are treated in this way, as well as regular files and directories, although there are inevitably some variations for each type. A file system object can be referenced from more than one entry in the name space - each entry (*link*) is of equal importance, and the object is not removed until all its links have been deleted and all its *descriptors* (see below) closed.

Opened files are accessed through *descriptors,* which are kernel structures referenced from an array in the process's *u* structure. System calls use indices into this array to identify descriptors. By convention, the first three file descriptors are reserved for input (*stdin*), output (*stdout*), and diagnostic output (*stderr*). Like most other process attributes, file descriptors remain valid across fork() and exec()[1] calls, and this enables the invoker to determine which files or devices will be used on the standard i/o descriptors. In this way the same program be used in many different ways, for example -

- interactively, with a terminal device open on the three standard file descriptors
- as an asynchronous background process with input and output attached to regular files
- in a group of cooperating processes or pipeline, where the output from one process is passed as input to the next via *pipes*

[1] The programmer may set a close_on_exec flag in the descriptor to override this.

File system security is implemented through protection bits associated with each file. Each user is assigned a user ID, and may also be included in several system groups. Read, write, and execute permission bits are specified for the owner of each file, the members of the owning group of the file, and the remaining users excluded from the other categories. For directories, 'execute' permission is interpreted as search permission: the directory may be used in a pathname to reference a file in its sub-tree, but its contents are not visible to the user unless read permission is granted. Two further bits are specified for executable files: *set user ID (setuid)* and *set group ID (setgid)* . When either of these bits is set, any process which executes the file takes on the privileges of the file's owner or group owner (respectively). This mechanism is used to enable controlled user access to system files and other resources.

The user ID 0 denotes the *super-user* (also known as *root*), and carries special privileges including full access to all file system objects. A program which is "setuid root" is therefore a potential security hazard, and must be subject to close scrutiny before being installed in a system. Chapter 7 discusses a method of reducing the need for such programs.

I.4 Inter-process Communication

Pipes are used for passing information between processes. They are implemented in BSD systems through *sockets,* which provide a more general mechanism for remote and local inter-process communication. A socket is an end-point for communicating with a process, and can exist only while referenced by a descriptor in that process or one of its descendants[1]. If a socket has been bound to an address, other sockets can send messages to it or connect to it to form a byte stream with the same read/write interface as other file system objects. The address format and communications protocols available for a socket are determined by its specified *domain* - for processes resident on the same machine the *UNIX domain* is normally used, and socket addresses are entries in the file system name space.

[1] Descriptors can be passed to other processes through UNIX domain sockets. A socket can therefore be kept open by an unrelated process if this unusual feature has been employed.

I.5 Terminal Devices

[For further information on the devices covered in this section, the reader is directed to the 'tty' and 'pty' entries in section four of the 4.3BSD programmers' manual. A degree of familiarity with these devices is assumed in Chapter 6.]

The terminal device driver is one of the least elegant aspects of modern UNIX systems. The original design suggests orientation towards hard copy terminals, and over the years the same mechanism has been extended to take advantage of more sophisticated CRT devices and now multi-windowing terminals. At the same time, the responsibilities of the driver have been extended to include such things as the generation of BSD job control interrupts. In fact, BSD systems separate the newer functions into a new *line discipline*, which is set when the user is working with a job-controlling shell. I/O control (*ioctl()*) calls are used to perform miscellaneous device driver functions including the setting and querying of parameters, and there are now over 30 defined ioctl() calls and 50 parameters for the terminal driver.

One of the problems associated with the terminal driver is that some of its attributes are more properly associated with a process rather than with the terminal itself. This is most noticeable when working with job controlling systems. Interactive applications often reset the terminal mode to allow immediate processing of typed characters or to disable echoing; if the process is killed or suspended, the terminal must first be restored to its original state and it must be re-configured if the process is subsequently resumed. The incremental nature of the terminal driver's development has resulted in four separate device parameter structures, and it is often necessary to reset all of these when control passes between applications.

Some programs attempt to format their output according to the perceived nature of the connected file or device; such semi-determinate behaviour can cause apoplexy amongst UNIX users when the heuristics fail, as they invariably do. For interactive applications which rely on the ability to manipulate the terminal driver directly, however, it is necessary to ensure that the input and output devices behave as terminals. When a filter process must be attached to such an application - e.g. as a multiplexor stage in a windowing system - standard inter-process communication facilities break down.

To counteract this problem, BSD UNIX has introduced special devices called *pseudo-terminals* (*ptys*), to assist communication with processes which expect to

be connected directly to a terminal. Each pseudo-terminal comprises two devices, master and slave. The slave side has an interface identical to that of a terminal device. Anything written to the slave device appears as output to the master device and vice-versa. This enhancement is particularly important for windowing terminals, as a window server process can fork an independent session with its own slave for each window, holding the master in order to direct i/o to the appropriate window on the terminal.

More general solutions to the problems of device-dependent i/o at application level have been suggested in the STREAMS packages of recent AT&T systems, and in the ethereal "descriptor wrapping" facilities promised in recent BSD releases [UCB 86a].

Appendix II

Project Library Routines

II.1 File Handling Package

```
#include <jhfs.h>

typedef struct fileid {
    int pathref;                /* path reference number of open object */
    int fflags;                 /* see below */
    fnamebuf fname;             /* final component of object's pathname */
    char fkind;                 /* K_FILE, K_VOL or K_DIR - see below */
    xparamblockrec pbp;         /* record obtained from pbget{vol,f,cat}info() */
    } fileid;

fileid *
makefid(gfidp, pathname, vwrefnum, dirid, findex, expkind, fflags)
fileid *gfidp;              /* fileid storage; NULL if automatic allocation desired */
char *pathname;             /* vol/file/dir name, (partial) pathname, or NULL */
int vwrefnum;               /* volume or working directory reference number, or 0 */
long dirid;                 /* directory ID or 0 */
int findex;                 /* file/directory index number or 0 */
int expkind;                /* (expected) file system object type */
int fflags;                 /* flags as specified below */
```

makefid() is the core routine of the fileid system. Given a combination of Macintosh file system object identifiers (name/pathname, volume/working directory reference number, directory ID, index number), it determines whether the object can be identified umambiguously, and if so performs the operation requested in the fflags parameter. If the operation is successful, a fileid structure containing all relevant information about the object is returned, otherwise NULL is returned and a suitable alert message is posted.

It is an error if any of the supplied identifiers conflict, or if the information given is insufficient to identify the object uniquely, but note that the current working volume or directory may be applied if the pathname is incomplete and vwrefnum is given as 0. A special reference number defined as IMPOSSVREF can be passed in vwrefnum to override this. If an object is specified by full pathname and the volume is not mounted, makefid() calls jdiskreq() to allow the user to insert the appropriate disk.

The expkind parameter is used to limit the validity of the operation to a given class of objects. The classes are denoted by the flags K_FILE, K_DIR and K_VOL; these flags may be added if the class of the object is unknown, so that a call specifying (K_VOL + K_DIR) will be valid if the object is a volume or a directory.

(K_VOL + K_DIR + K_FILE) is pre-defined as K_UNK. The class K_DIR is valid for HFS volumes only, and only if the 128K ROM File Manager is installed. An error is raised if the object exists but is not of the requested class.

Interpretation of fflags:

The following flags cause the specified object (file or directory) to be opened; the path reference number is returned in the fileid structure. O_RDONLY positions the file pointer at the start of the file. Write modes cause the file pointer to be set to the logical end-of-file unless the O_TRUNC flag is also specified, in which case the file is truncated to zero length.

If the object is a directory, O_RDWR opens a working directory; no other modes are valid. When the application exits, all working directories should be closed by calling hfsexit().

If the request is for a resource file, the permissions are only valid if the 128K ROM File Manager is installed. O_SHRDWR is never valid for resource files.

O_RDONLY .....open for reading only
O_WRONLY.....open for writing only
O_RDWR.....open for reading and writing
O_SHRDWR.....open in shared write mode

The supplementary open flags apply only to files:

O_TRUNC.....truncate the file before writing.
O_DATAF.....open the data fork of the file
O_RESFL .....open the file as a resource file
O_RAWRF.....open the raw resource fork of the file
O_TXTDF.....open the data fork of the file, and set a flag for line separator translation in fidread() and fidwrite().

The following flags cause the specified object (file or directory) to be created if it does not already exist.

F_CREAT.....create the last component of the pathname iff its parent exists
P_CREAT.....create any directories needed to complete the path as well
F_ERREX.....raise an error if the object already exists

The flags passed to makefid() are included in the fileid structure. F_ERREX, however, is not passed on and the flag is overloaded with F_NEW, which is set if the file was created in the makefid() call; this notifies the application to set the Finder information.

If no flags are specified, makefid() simply checks for the existence of the object, and returns the object information in the fileid.

```
openfid(fid, fflags)
fileid *fid;        /* object to be opened (file or directory) */
int fflags;         /* open flags */
```

openfid() is used to open the object specified by an existing fileid structure. It is used when the fileid comes from a source other than makefid(), or when the application must test the creator and/or type of a file before opening it.

```
jsetwd(fname, wvrefnum, wddirid)
char *fname;        /* directory name, or file pathname */
int wvrefnum;       /* working directory / volume reference number */
long wddirid;       /* directory id of new current working directory */
```

jsetwd() sets the current working directory and/or volume to that specified by the parameters. If a file name is given, the new working directory is the parent of the file.

```
rscandir(dfidp, fftn, sdftn)
fileid *dfidp;      /* volume or directory fileid */
procptr (*fftn)();  /* handler for each contained file */
procptr (*sdftn)();     /* handler for each contained sub-directory */
```

rscandir() scans the files and subdirectories contained in a given volume or directory, creating fileid's for each and calling fftn(subfileid, parentid) for each file and sdftn(subdirid, parentid) for each subdirectory. HFS volumes are treated as root directories. The arguments to fftn() and sdftn() are initialised fileid's. Recursive scans can be implemented by calling rscandir() indirectly through sdftn().

```
int
fidread(fdp, buf, count)
fileid *fdp;        /* file opened for reading */
char *buf;          /* buffer for characters to be read from file */
int count;          /* number of characters to be read */

int
fidwrite(fdp, buf, count)
fileid *fdp;        /* file opened for writing */
char *buf;          /* buffer of characters to be written to file */
int count;          /* number of characters to be written */
```

fidread() and fidwrite() provide Unix-like read and write routines for use with fileid structures. If the file is opened with the O_TXTDF flag set (text data fork), fidread() converts carriage returns to linefeeds and fidwrite converts linefeeds to

carriage returns. This feature reflects the different line separator conventions of Unix and the Macintosh. The number of characters successfully read/written is returned. The routines guarantee that the requested bytes will be read/written unless a file system error occurs or end-of-file is reached (fidread() only).

```
int
fserrhdl(errcode, ckind, pname)
int errcode;        /* file system error code */
int ckind;          /* file object type - file, volume or directory */
char *pname;        /* object name and/or user message */
```

fserrhdl() displays an alert box explaining a given file system error code. The object type is K_FILE, K_DIR, K_VOL or K_UNK as defined in the fileid system.

```
int
scanpath(pname)
char *pname;            /* pathname to be interpreted */
```

scanpath() checks the validity of the given pathname, and returns the class (K_FILE, K_DIR, K_VOL) of the object which it specifies. If class cannot be determined from the name, a combined class (e.g. K_FILE + K_DIR) is returned. If the name is invalid, scanpath() returns (-1).

```
correctname(fkind, sname, dname)
int fkind;          /* object class - K_FILE, K_DIR or K_VOL */
char *sname;        /* source pathname */
char *dname;        /* destination object name buffer */
```

correctname() takes a pathname and an object type, extracts the appropriate component of the path (volume name, lowest-level directory name, or file name) and copies it to the destination with colons placed according to the rules indicated in the File Manager documentation - i.e. "vol:", ":dir:", "file".

```
deunlxlfy(pname)
char *pname;        /* Unix pathname, returned as Mac. equivalent */
```

deunixify() takes a Unix-style pathname and translates it into the equivalent Macintosh pathname, reversing the path separators of the two systems.

```
int
rftypeauth(rfid, rtype, rauth)
fileid *rfid;    /* target file */
char *rtype;  /* file type ID */
char *rauth;  /* file creator ID */
```

rftypeauth() checks or sets the type and author of the file identified by rfid. If the fileid has the F_NEW flag set, the values are set; otherwise the existing values are compared against the parameters, and an alert is posted if they differ. If the call succeeds (i.e. the set succeeds or the values are identical) 0 is returned; if not, -1 is returned.

jhfsexit()

jhfsexit() is called before an application exits. It closes any working directories which the application has opened.

```
int
jdiskreq(vname)
char *vname;          /* name of disk to be mounted */
```

jdiskreq() posts a modal dialog requesting that the named disk be inserted, after ejecting the internal disk if necessary. The dialog waits for a disk-insertion event or a cancellation. If a disk is inserted, it is mounted and its name compared with that of the requsted disk; if the names are identical the routine returns 0 (noerr), otherwise the process is repeated. The user may press the dialog's cancel button if the volume is unavailable, in which case the routine returns the file system error code nsverr (no such volume).

The fileid system includes a number of boolean C preprocessor macros for obtaining file system information from initialised parameter block structures (as contained in a fileid) and for identifying reference numbers:

```
hfsok()                - the 128K ROM file manager is installed

ishfsvol(vpb)    - the volume is HFS-initialised
volumeparam vpb;

catisdir(cpb)    - the object is a directory
cat{file,dir}param cpb;

catisfile(cpb)   - the object is a file
cat{file,dir}param cpb;

catisroot(cpb)   - the object is a root directory
cat{file,dir}param cpb;

isdrvnum(x)      - the reference number is a drive number
int x;

isvolref(x)      - the reference number is a volume refnum
int x;

iswdref(x)       - the reference number is a working directory refnum
int x;
```

II.2 Support Routines for Other File Handling Systems

```
FILE *
mfopen(fname, vwref, pdir, omode)
char *fname;      /* file name */
int vwref;        /* volume/working directory reference number */
long pdir;        /* parent directory ID */
char *omode;      /* stdio fopen() mode */
```

mfopen() gives access to Megamax's buffered stdio routines, without forcing the programmer to re-create the full pathname of the target file.

```
settypauth(fname, fvref, ftype, fauth)
char *fname;      /* file name */
int fvref;        /* volume/working directory reference number */
char *ftype;      /* file type ID */
char *fauth;      /* file creator ID */
```

settypauth() sets the type and author fields in the specified file.

```
int
sfgf(fname, fvref, msg, types, filterproc)
char **fname;          /* returned file name */
int *fvref;            /* returned volume / working directory refnum */
char *msg;             /* sfgetfile() message */
char *types;           /* string of file types - eg "APPLFNDR" */
procptr filterproc;    /* file filter */
```

sfgf is an alternative interface to sfgetfile(). It returns TRUE if a file is chosen, otherwise FALSE.

```
int
fidtosfr(fidp, replyp)
fileid *fidp;       /* supplied fileid */
sfreply *replyp;    /* returned standard file reply structure */

int
sfrtofid(replyp, fidp, fflags)
sfreply *replyp;    /* supplied standard file reply structure */
fileid *fidp;       /* returned fileid - must be pre-allocated */
int fflags;         /* makefid() flags */
```

fidtosfr() and sfrtofid() convert between fileid and standard file reply structures.

```
long
vfreebytes(vref)
int vref;       /* reference number of target volume */
```

vfreebytes() returns the number of bytes which are available for file allocation on the given volume - i.e. (free blocks * block size). Note that on HFS volumes, the existence of N free blocks is not always a guarantee that a file of N blocks will fit on the volume, since extra blocks may have to be allocated for the category and extents tree files.

II.3 Dialog Support Routines

```
typedef struct stdditem {
    handle ihdl;            /* item handle - used internally */
    rect irect;             /* item rectangle - used internally */
    char itype;             /* item type - used internally */
    char iflag;             /* item flags - see below */
    union {
        procptr iu_btnact;          /* button action procedure */
        procptr iu_usract;          /* user item defn. proc. */
        stringhandle iu_tstr;       /* text item string */
        int iu_cbval;               /* check box value */
        int *iu_rditmp;             /* radio button group */
        long iu_numval;             /* numeric text item value */
        } itmu;
    } stdditem;

#define STDFILTER       ((procptr)(-1))
#define BTNFINISH       ((procptr)NIL)
#define UIDFT           ((procptr)NIL)
#define EDNUMERIC       1
#define EDTEXTSEL       2
#define EDCHANGED       4
#define CTLINACTV       1

int
stddialog(dlgid, itma, nitems, fproc)
int dlgid;          /* dialog template & item list resource ID */
stdditem *itma;     /* item definition structure list */
int nitems;         /* number of items in itma */
procptr fproc;      /* event filter function */
```

stddialog() is a simplified method for implementing modal dialogs. An array of initialised stdditem records, one for each item in the dialog, is passed as a parameter. stddialog() displays the dialog and retains control until the user interaction is complete, whereupon it removes the dialog and returns the item number which terminated the session.

The stdditem records are used to determine the responses to ModalDialog()'s result codes. The order of the records must correspond to the order in which the items appear in the Dialog Template i.e. the first record (itma[0]) corresponds to item 1 and so on.

The calling routine need only initialise two fields of each stdditem structure - iflag and itmu. The values depend on the type and usage of the dialog item:

user items: itmu contains a definition procedure. If this is given as UIDFT, the default button outlining routine is installed. User items are treated as disabled.

control items (buttons, check boxes, radio buttons): if CTLINACTV is specified in the iflag field, the control is greyed out (made inactive). Control settings are updated automatically within stddialog(). The use of control templates is not supported.

buttons: itmu contains an action procedure. If BTNFINISH is specified here, pressing the button causes stddialog to return. The default button outlining routine and the standard filter function expect the first item to be the default button. If no default button is desired, these routines should not be used, or the first item should be a hidden button.

check boxes: itmu contains the initial value of the box - 0 or 1 as defined in the control manager. The value is updated according to the user's actions, and can be inspected when stddialog() returns.

radio buttons: itmu contains a pointer to an integer variable set up by the caller. All radio buttons in a group should reference the same variable, and the variable should be set to the item number of the radio button which is to be selected initially. The item number in the variable is updated according to the user's actions, and can be inspected when stddialog() returns. In accordance with the guidelines, only one radio button in a group can be selected at a time.

static text: itmu may contain NIL if the text is defined in the item list resource, or it may contain a toolbox string handle to override the resource value.

editable text: itmu contains a toolbox string handle. If the handle is empty, no default text will be displayed, otherwise the edit box will be initialised to the given string. One text box should have EDTEXTSEL set in the iflag field, which causes text in the box to be selected when the dialog is first displayed. If a given box is edited by the user, EDCHANGED will be set in its iflag field when stddialog returns, and itmu will contain the new string handle.

editable numbers: If an editable text item is to contain integer values only, EDNUMERIC should be set in iflag, and itmu should contain the initial value (as a long integer). The standard filter ignores attempts to type non-numeric characters into the box. The new value is returned in the itmu field as a long integer.

An alternative filter procedure can be specified for ModalDialog. The standard filter procedure (specified by STDFILTER) supports the use of the editing key equivalents, the default button keys, and the interpretation of numeric edit items.

```
int
addditem(dlgp, itype, irect, idata)
dialogpeek dlgp;        /* existing dialog window */
int itype;              /* item type; may include the "itemdisabled" flag */
rect *irect;            /* item's display rectangle */
(variable) idata;       /* type depends on itype */
```

addditem() adds a dialog item to an existing (displayed) dialog, returning the item number of the new item. idata takes the following values:

itype:	idata:
useritem	procptr - useritem definition procedure
ctrlitem+resctrl	int - resource id of control template
ctrlitem+(other)	ptr - title of standard control
picitem	int - resource id of picture
iconitem	int - resource id of icon
stattext, edittext	ptr - initial item text

```
delditem(dlgp, itmid)
dialogpeek dlgp;  /* existing dialog window */
int itmid;        /* item number of item to be deleted */
```

delditem() deletes a specified dialog item from an existing dialog.

```
int
savedtempl(dlgp, dlgresid)
dialogpeek dlgp;  /* dialog to be saved */
int dlgresid;     /* resource ID of dialog template and item list */
```

savedtempl() saves the existing dialog referenced by dlgp as a dialog template resource and a dialog item list resource. There is a restriction on resctrl-type items: the control template ID is not directly obtainable from the loaded dialog structure; applications using these items must use other methods ensure that the appropriate ID is installed. If the save is successful, -1 is returned, otherwise 0.

```
embolden(dlgp, dfti)
dialogptr dlgp;   /* target dialog */
int dfti;         /* item number of default button user-item */
```

embolden() installs a button highlighting definition procedure in the specified dialog. The procedure assumes that the default button is the first item in the dialog, and that the given item number has been declared as a user item.

```
setuitem(dlgp, inum, uproc)
dialogptr dlgp;   /* target dialog */
int inum;         /* item number of user-item */
procptr uproc;    /* user item definition procedure */
```

setuitem() installs a definition procedure for a disabled user item

```
int
FindDItem(dlgp, hpt)
dialogpeek dlgp;  /* target dialog */
point *hpt;       /* mouse click location */

HideDItem(dlgp, itmn)
dialogpeek dlgp;  /* target dialog */
int itmn;         /* item to be hidden */

ShowDItem(dlgp, itmn)
dialogpeek dlgp;  /* target dialog */
int itmn;         /* item to be shown */
```

FindDItem(), HideDItem() and ShowDItem() are compatibility routines for 64K ROM machines. Refer to Inside Macintosh IV.

II.4 Resource Management Routines

```
handle
addnewres(dataptr, datalen, rtype, rid, rfile)
char *dataptr;    /* resource data */
long datalen;     /* length of resource data in bytes */
char *rtype;      /* resource type */
int rid;          /* resource id */
int rfile;        /* target resource file */
```

addnewres() adds a new resource to a specified resource file, replacing any existing resource with the same type and ID. Possible errors are checked and alerted if necessary. addnewres() ensures that the resource is written to the file. curresfile() is not changed. A handle to the new resource is returned (NULL if the routine fails).

```
handle
getormakeres(rtype, rid, rname, minsize)
char *rtype;      /* resource type */
int rid;          /* resource id */
char *rname;      /* resource name - for creation only */
long minsize;     /* minimum size for resource */
```

getormakeres() returns the requested resource (zero-extended to minsize if necessary) if it exists in the current resource file. Otherwise it creates and returns a new resource of minsize bytes in the current resource file. If an error occurs, an alert is displayed and NULL is returned.

```
handle
getresexcurrf(rtype, rid)
char *rtype;      /* resource type */
int rid;          /* resource id */
```

getresexcurrf() returns the requested resource if and only if it is found in the current resource file.

```
int
Incurrf(rtype, rid)
char *rtype;  /* resource type */
int rid;      /* resource id */
```

incurrf() returns TRUE iff the current resource file contains an instance of the specified resource. The resource is not loaded into memory.

```
int
saveresource(reshdl)
handle reshdl;    /* resource to be written out */
```

saveresource() forces the given resource to be written to its resource file. If the write fails, an error alert is posted and a Resource Manager error code is returned, otherwise the routine returns 0.

II.5 Error Handling

```
#define mesalrtid      463

sysalrterr(estr, args)
char *estr;    /* printf() format string */
int args;      /* printf() arguments */
```

sysalrterr() posts an alert box which displays the given arguments after formatting by sprintf(). The alert box template and item list have resource ID 463 and are supplied in the resource file which accompanies the library. sysalrterr() uses a fake return to pass its arguments direcuy to sprintf().

The alert box contains three other fields which are filled by sysalrterr(). These are for :

- the global variable errno, which is set by all library routines in which toolbox calls may fail.
- the return address for sysalrterr()
- the global string pointer sysalrtloc, which can be set by the programmer.

The alert can be dismissed by one of two buttons:

- the 'OK' button, which causes execution to continue
- the 'debug' button, which invokes the system debugger

The supplementary fields and the debug button should hidden (using ResEdit) when the program is released for general use.

```
setabort()                  /* C preprocessor macro */
abortjmp(abortcode)         /* C preprocessor macro */
```

setabort() sets up a jump buffer to be used as a return location during error recovery. abortjmp() performs a non-local goto to the most recently established setabort() buffer. setjmp() returns 0 on the first call, and an integer error code passed to abortjmp() when returning from an error.

```
int
getabortcmd()
```

getabortcmd() returns TRUE if the user has pressed ⌘-. to cancel a long operation.

```
checkabort()
#define aborterr 1978
```

checkabort() calls abortjmp(aborterr) if the user has pressed ⌘-. to cancel a long operation.

II.6 Memory Management

```
alloca(nbytes)
int nbytes;         /* number of bytes to be allocated; must be even */
```

alloca() allocates temprory storage in the current stack frame. The allocated area may be altered subsequently by calling alloca() with a positive or negative argument. The area is invalidated when the calling routine returns. N.B. If the calling routine uses any register variables, the area must be deallocated explicitly ("alloca(-nbytes)") before the routine returns.

```
char *
allocstr(str)
char *str;          /* string to be duplicated */
```

allocstr() copies a given C string to a newly allocated area on the heap. A pointer to the new string is returned. If no memory is avalailable, the error is alerted and NULL is returned.

```
char *
mmalloc(nbytes)
int nbytes;         /* number of bytes to be allocated */
```

mmalloc() calls malloc(), and posts an alert if the memory is not available.

```
bzero(start, len)
ptr start;      /* block to be cleared; must be on a word-boundary */
int len;        /* length of block in bytes; need not be even */
```

bzero() uses a fast assembly loop to clear the given block. Longword instructions are used until less than four bytes remain.

The following memory management C pre-processor macros are defined in <stdc.h>:

cnewp(x)	return a pointer to a new heap object of type x
cnewa(x, y)	return a new heap array containing y objects of type x
cfree(x)	free the object pointed to by x and clear x
bcopy(x, y, z)	block copy z bytes from pointer x to pointer y - NB Megamax C supports structure assignment
clearstruct(x)	clear structure x
clearstp(x)	clear structure pointed to by x

II.7 String Manipulation

```
cvtescapes(s)
char *s;      /* string to be converted */
```

cvtescapes() converts backslash-marked non-printing characters in the given string according to the conventions of the C scanner:

\n.....linefeed

\r.....carriage return

\t.....horizontal tab

\b.....backspace

\f.....formfeed

\E.....escape

\0.....null - note that this will terminate the string!

\xxx.....the given octal ascii sequence

\\.....backslash

In all other cases the backslash is removed from the string. Note that the string cannot increase in length, so it is safe to perform the conversion in situ.

```
char *
striphead(dstp, srcp, brkch, len)
char *dstp;     /* destination string */
char *srcp;     /* source string */
char brkch;     /* separator character */
int len;        /* maximum length of destination  string, excluding terminating '\0' */
```

striphead() copies the source string to the destination string, up to but excluding the first occurrence of brkch. Returns the first position in the source string containing brkch. If brkch is not found, the string is copied completely and the

position of the terminating '\0' of the source string is returned, unless the source string is longer than len in which case only the first len characters are copied and the position returned is &srcp[len].

```
char *
striptail(dstp, srcp, brkch, len)
char *dstp;        /* destination string */
char *srcp;        /* source string */
char brkch;        /* separator character */
int len;           /* maximum length of destination  string, excluding terminating '\0' */
```

striptail() is identical to striphead, except that the last occurrence of brkch is used rather than the first.

```
pascpy(dst, src)
char *dst;    /* destination Pascal string buffer */
char *src;    /* source Pascal string */
```

pascpy() copies a Pascal string from the source to the destination.

```
int
addIndString(astr, istrhdl)
char *astr;              /* C string to be added to toolbox string list */
stringlist **istrhdl;    /* target string list */
```

addIndString() adds the given C string to the end of the given toolbox (Pascal) string list.

```
mergestring(strlid, newstr, strnum)
int strlid;        /* resource ID of string list */
char *newstr;      /* replacement C string */
int strnum;        /* string index */
```

mergestring() replaces the Nth string in a given toolbox string list with the given C string. If the supplied index number is greater than the number of existing strings, the list is padded with null strings to make newstr the Nth and final string.

II.8 Interface Graphics

```
int
dragrect(therect, startpt, bdsrect)
rect *therect;     /* rectangle to be dragged - returned with translation */
point *startpt;    /* initial mousedown location (global coordinates) */
rect *bdsrect;     /* bounds rectangle - c.f. dragwindow() */
```

dragrect() drags a grey outline of the given rectangle within bdsrect. If the rectangle is moved from its original location, the translation is applied to *therect and the routine returns TRUE; otherwise it returns FALSE.

```
long
growrect(therect, startpt, bdsrect)
rect *therect;     /* rectangle to be re-sized */
point *startpt;    /* initial mousedown location (global coordinates) */
rect *bdsrect;     /* bounds rectangle - c.f. growwindow() */
```

growrect() tracks the mouse, drawing a grey outline of the given rectangle as it is re-sized. The top left corner of the rectangle is pinned. The final size change is encoded in a longword according to the same convention as growwindow(). growrect() should be called with an initial mouse location near the bottom right hand corner of the rectangle; any initial offset is taken into account when drawing the outline and in the final size calculation.

```
int
Incorner(ptp, rectp)
point *ptp;        /* mousedown location */
rect *rectp;       /* target rectangle */
```

incorner() returns TRUE iff the given point is contained within a 5x5 pixel square in the bottom right corner of the given rectangle. It is intended for use with growrect().

```
setslop(slopr, limr)
rect slopr;    /* slop rectangle */
rect limr;     /* limit rectangle */
```

setslop() sets the slop rectangle 10 pixels out from the limit rectangle for dragging operations (ref. Control Manager Documentation).

II.9 Queues

```
typedef struct jqel {
    struct jqel *next,
                *prev;
    struct queue *q;
    int body[0];        /* convenient fiction - actual contents will vary */
    } jqel;

typedef struct queue {
    struct jqel *head,
                *tail;
  } queue;

enqueue(pp, qp)
jqel *pp;              /* element to be inserted */
queue *qp;             /* target queue */
```

enqueue() inserts the given element at the tail of the specified queue

```
unqueue(pp)
jqel *pp;              /* element to be removed */
```

unqueue() removes the the given element from the queue into which it has been linked, if any.

II.10 Set Manipulation

```
#include <varargs.h>

typedef int *set;

set
makeset(va_alist)
va_dcl
```

makeset() returns an array containing the (integer) values specified in the argument list. A version of the portable C library variable length argument list mechanism is defined in varargs.h. The first argument and the first element in the array is the number of elements in the set. Sets may be deallocated by free().

```
int
member(tstval, cset)
int tstval;     /* element to be tested for set inclusion */
set cset;       /* target set defined by makeset() */
```

member() returns TRUE if the given element is a member of the given set.

II.11 Miscellaneous Routines

```
osinit(exitproc)
procptr exitproc; /* ResumeProc for initdialogs() */
```

osinit() initialises QuickDraw, the Font Manager, TextEdit,and the Dialog Manager. It installs the given resume procedure for D.S. error recovery, and also loads the general alert template for sysalrterr().

```
getargs(argcp, argvp, vravp, typavp, mesgp)
int *argcp;         /* cardinality of argument identification lists */
char ***argvp;      /* internally allocated array of file name pointers */
int **vravp;        /* internally allocated array of volume/wd refnums */
long **typavp;      /* internally allocated array of file types */
int *mesgp;         /* print flag (Ref. Inside Mac) */
```

getargs() returns the application parameters as supplied by the Finder and other application launchers (BatchX, macscrewn). The first application parameter returned refers to the application itself, following the C convention. All vectors are null/zero terminated.

```
int
isdblclk(ev2)
eventrecord *ev2;       /* mousedown event (all other events are ignored) */
```

isdblclk() should be called with each mousedown event that occurs while double clicks are significant. It returns TRUE if the previous mousedown event occurred within 5 pixels of the current event, and if the time between the two events is less than the value returned by getdbltime().

The Interface Guidelines suggest that double clicks should be determined by the time difference between a mousedown and the preceding mouseup. I consider this behaviour undesirable; when the user drags an item, but returns to the original location and quickly reselects, a double click will be reported. Many other applications follow the method applied in this routine, which I consider preferable.

A number of general purpose C preprocessor macros are defined in stdc.h:

MIN(x,y)	minimum (more negative) of x and y
MAX(x,y)	maximum (more positive) of x and y
ABS(x)	absolute value of x
ABSDIFF(x,y)	absolute value of (x - y)
SIGN(x)	1, 0 ,-1 for x positive, 0, or negative
hibyte(x)	the most significant byte of int x, as an int
lobyte(x)	the least significant byte of int x, as an int
hiword(x)	the most significant word of long x, as an int
loword(x)	the least significant word of long x, as an int
RESPTR(rhdl,ptyp)	the pointer to the object of type ptyp which is stored in the resource handle rhdl
mglobal(adr, typ)	the object of type typ which is stored at the absolute address adr (Value is an lval - used for referencing Macintosh low memory globals)
newrom()	true if the 128K ROM is installed
debug()	invoke the system debugger

Appendix III

Difficulties in MPFW Process Implementation

III.1 Shared System Globals

There is a misconception abroad[1] that the lack of pre-emptive multitasking on the Macintosh is the result of some hardware limitation in the MC68000. It is true that some 68000 instructions cannot be restarted reliably after bus and address errors, and although this hampers the use of paged virtual memory, it is of no consequence in implementing pre-emptive multitasking.

The problems arise from the design of the Macintosh toolbox and operating system. Many structures which in a multitasking system would apply only to a single process (e.g. the current port, resource file, and working directory, and a host of toolbox state variables) are stored globally - that is, in the low-memory globals and system heap, rather than in the application space. It is nevertheless possible to enumerate these and swap them to private areas during process exchange, and presumably this has been done in programs such as SWITCHER and MULTIFINDER.

MPFW does not need to handle as many structures as full-application switchers. From the system's point of view it is a single stand-alone application, and many potential conflicts (desk accessory handling, menu bar and screen sharing, event distribution etc.) do not arise because the sub-applications use the high-level services of MPFW and do not maintain individual system-level structures (e.g. they do not initialise the toolbox managers). Nevertheless, the process dispatcher is required to save a number of global variables in the stack frame during process exchange.

III.2 Non-Reentrant System Code

A more serious difficulty is in determining whether interrupt-driven (pre-emptive) process exchange could leave the shared OS and Toolbox structures in an inconsistent state. Normally this is done by ensuring that such inconsistencies can arise only during the execution of a single system call, and (in 68000 machines) by

[1] held even by some Apple employees - USENET article <7670@apple.Apple.Com>, March 1988.

setting the supervisor mode bit (S-bit) in the processor status register during system calls. This switches the processor from user (unprivileged) mode to supervisor (privileged) mode The process dispatcher is inhibited when the S-bit is set. In fact, this function of the S-bit is of secondary importance in systems with hardware memory management; the primary purpose is to control access to protected memory areas containing system structures and other processes' data.

Unfortunately, the Macintosh always keeps the processor in supervisor mode. Without a more detailed knowledge of the operating system implementation it is difficult to determine whether this is strictly necessary, but it appears that the saved status register may be restored immediately after some traps and this would foil any attempts to employ the status bit for MPFW's purposes. The Macintosh guidelines prohibit user mode programs, but they also prohibit the use of privileged instructions which are required in any case; the side-effects of running MPFW code in user mode are unknown.

Instead, MPFW simulates the simulate the S-bit functionality in software, using the *ApplScratch* global space as a guaranteed "known" location. The global flag *CANTXFER* disables the process dispatcher, and is set during trap calls and also during some operations in MPFW itself. For trap calls, MPFW must replace the A-line trap dispatcher with its own handler, which ensures that the flag is set for the duration of the call.

This handler, although only 21 instructions long, was one of the most difficult routines in the project. Because most debuggers also intercept the A-line trap dispatch vector it was unwise to replace the dispatcher completely, so the new handler had to arrange for prologue and epilogue operations around the original handler. The 68000 trap exception stack frame contains the address of the instruction which caused the trap, and the status register. Normally, the A-line trap dispatcher uses the exception address to find the trap word, which in turn gives an index for the appropriate routine in the dispatch table (§3.2.1). The main problem is to find a method of returning to the epilogue routine while leaving the trap word visible, and without changing the offset from the stack pointer to the call parameters.

MPFW's A-line trap interception works as follows: the prologue sets the flag word, and then replaces the exception address on the stack with the address of the

epilogue. The original exception address is incremented (to give the required return address) and saved in a low-memory global. Before jumping to the trap dispatcher, the prologue replaces the first instruction of the epilogue with the trap word found at the original exception address. In this way, it creates the illusion that the epilogue is actually the trap caller.

Except for one small point. Scott Knaster [KNASTER 86] reveals that certain ROM patches use the absolute value of the exception address to determine whether they were called from a given ROM routine. In this way a short ROM routine can implement a patch for a much longer one which calls it. MPFW's handler gets around this by checking the value of *CANTXFER* before performing the above actions; if it is already set - indicating that a ROM routine is already active - it jumps straight to the trap dispatcher.

Fig. III.1 Trap Dispatcher Interception Routine

```
                Code                                Stack format & comments
                                                    /*  <oldTOS> <4EA> <2SR> */
prologue:       tst.b   CANTXFER                    /*  jump directly if already set */
                beq.s   modify                      /*  otherwise we have a job to do */
                jmp     0xFFFFF                     /*  -> jmp oldtrapdisp.l  */
modify:         addq.b  #1, CANTXFER                /*  quicker than a move */
                movem.l A0/A1, -(A7)                /*  <oldTOS> <4EA> <2SR> <8>*/
                movea.l 10(A7), A0                  /*  EA -> A0 */
                lea     epilogue, A1                /*  &epilogue -> A1 */
                move.w  (A0), (A1)                  /*  A9XX -> epilogue[0] */
                addq.l  #2, A0                      /*  RA -> A0 */
                move.l  A0, SAVERA                  /*  RA -> global */
                move.l  A1, 10(A7)                  /*  <oldTOS><4MYEA><2SR><8> */
                movem.l (A7)+, A0/A1                /*  <oldTOS><4MYEA><2SR> */
                jmp     0xFFFFF                     /*  -> jmp oldtrapdisp.l */

epilogue:       dc.w    0xA000                      /*  dummy trap copied from code */
                move.l  SAVERA, -(A7)               /*  restore saved return address */
                move.l  SR, -(A7)                   /*  push trap routine status */
                                                    /*  assert : CANTXFER == 1 */
                clr.b   CANTXFER                    /*  clear flag */
                tst.b   XFERPEND                    /*  set if process xfer was blocked */
                beq.s   immrtn                      /*  if not, return now */
                jmp     0xFFFFF                     /*  -> jmp runproc; runproc rte's */
immrtn:         rte                                 /*  return & restore trap routine cc's */
```

If the process dispatcher is invoked during the execution of the trap, it sets another flag *XFERPEND* to indicate that exchange was blocked. After the trap call return, the epilogue creates a fake exception stack frame using the stored return

address and the current status register (as returned by the trap[1]). It then clears *CANTXFER*, working on the assumption that *CANTXFER* must have been zero in the prologue. If *XFERPEND* is set, the epilogue jumps to the process dispatcher to allow the exchange to proceed - the faked exception stack frame appears identical to that produced by a VIA interrupt (ref §III.1.4). Otherwise, the epilogue performs an exception return (RTE) back to the calling routine.

III.3 Asynchronous Process Exchange

The MPFW "scheduler" is implemented as a standard asynchronous VBLTask. It is an extremely simple piece of code; most of the work originally intended for it - notifying I/O processes, adjusting sub-application priorities etc. - is now performed in the main loop process, which is activated every $^1/_{60}$s in accordance with the requirements of *SystemTask()*. All the "scheduler" does is to drain[2] the main loop's idle semaphore (i.e. its event semaphore).

It is not possible to invoke the dispatcher from the VBLTask - this would cause a context switch in the middle of the system's interrupt-level tasks, and the results would be unpleasant. The context switch is therefore performed as an epilogue to the VIA interrupt handler. This is somewhat easier than in the case of the trap dispatcher: the routine simply increments *CANTXFER* to disable the dispatcher, executes the normal handler, decrements *CANTXFER*, and either jumps to the process dispatcher or RTE's to the interrupted code according to the values of *CANTXFER* and *XFERPEND*.

A cautionary note: the pseudo-code descriptions of MPFW's low-level routines (Fig. III.2) show a considerable amount of exception stack frame manipulation. This ensures that the process dispatcher (*runproc()*) always restores the status register (i.e. re-enables interrupts) with an RTE into the user code of the new context. Originally, the routines were much simpler: the semaphore routines and the trap dispatcher and VIA intercepts all transferred to the dispatch routine through a subroutine branch instruction. This left an extra stack frame which was not cleared

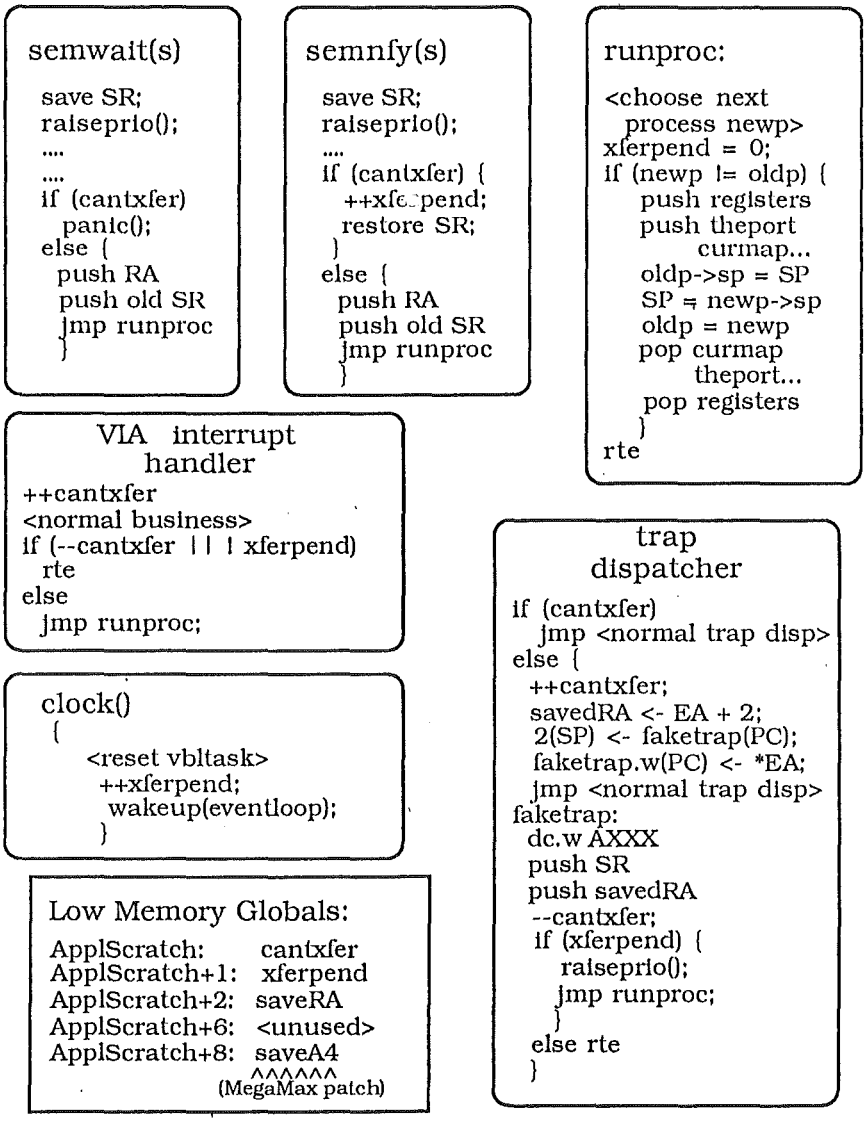
---

[1] At present nothing relies on the condition codes, but this is a safety measure.

[2] MPFW provides two extra semaphore primitives for system use in addition to the standard *wait* and *notify*. These are *semadd(process, semaphore)*, which installs a newly created process on a given semaphore, and *semdrain(semaphore)*, which reschedules all processes waiting on the semaphore and restores the initial count. The latter is used for non-critical wakeup operations, as above.

until after interrupts had been enabled. From time to time, a pathological condition would occur where the VIA interrupted as soon as interrupts were enabled, and this eventually caused stack overflow.

Fig. III.2 Low-level Routine Pseudo-Code



III.4 Stack and Heap Management

Application-level switchers (SWITCHER, MULTIFINDER) must allocate fixed-size application spaces. The disadvantage of this is that each application must specify its maximum and minimum memory requirements, and the maximum will be allocated if it is available. It would be preferable if the minimum could be allocated at first and expanded as necessary, thus allowing more applications to be launched simultaneously. The Macintosh memory model grants each application exclusive

access to a contiguous application area, however, and this is obviously not relocatable so fixed zones are necessary.

It was not seen as necessary to impose this restriction on MPFW sub-applications; while separate heap zones might have added some protection against cross-interference from dangling pointer references, efficient memory allocation was considered more important. All MPFW and sub-application structures therefore share the same heap, and the stacks are allocated as non-relocatable blocks within the heap. (The original stack belonging to the event loop process is retained, however.)

A third problem stems from this decision, and involves the Macintosh "stack-sniffer" routine. At every vertical retrace interrupt ($^1/_{60}$ s) the Macintosh checks that the stack pointer has not moved into the application heap area. It does this by comparing the value of stack pointer with the global *HeapEnd*. There is a mechanism for disabling this automatic check, but a less well-known fact is that several ROM routines use the same code to determine the amount of stack space available for block operations.

Officially, *HeapEnd* is defined as the upper limit of the application heap. Fortunately, however, it is used almost exclusively as above - i.e. to mark the lower limit of the stack area[1]. Most of the memory allocation routines use a field in the heap zone header to mark the end of the zone. It is therefore possible to set *HeapEnd* to the lowest address on the current stack; MPFW's process dispatcher does this. Nevertheless, there are some occasions when *HeapEnd* must be set to its correct location:

- *InitApplZone()* sets the value of *HeapEnd* when the application zone is first initialised; this happens at the start of the program and so does not concern us
- when Memory Manager is unable to meet a memory request, it attempts to expand the heap zone by calling *SetApplLimit()* . This increases the value of *HeapEnd* by a given amount up to the absolute limit specified in the global *ApplLimit*. To combat this, MPFW sets the heap zone to its maximum size at the start of the program.

[1]This information was obtained by searching the entire ROM and System File for references to the global's address, and disassembling the surrounding code. Subsequent communication with other developers involved in co-routine implementation confirmed that, on the basis of experience, the assumption was safe. The author knows of no-one else who risked permanent emotional disability by checking it, however.

- the call which allocates master pointer blocks, *MoreMasters()*, also checks *HeapEnd*. Currently, MPFW allocates a large number of blocks at the start of the program and hopes for the best. This is not really satisfactory, and a better mechanism would be for main loop process to determine the number of free master pointers and allocate them as required. Scott Knaster [KNASTER 86] recommends this even for ordinary applications, as a means of avoiding heap fragmentation.

III.5 Acknowledgement

I am indebted to Mr. G.C. Ewing for encouraging me to attempt pre-emptive multitasking on the Macintosh, and for providing me with sample core routines for a microprocessor multitasking operating system. This was one of his many contributions which, in the course of discussion, helped shape the overall design of MPFW.

Appendix IV

Interface Programming Language

IV.1 A Language for MPFW Sub-Applications

Given that the intention of MPFW was to provide a framework for users to implement and install their own interface sub-applications, it was necessary to provide users with a means of writing these and linking them into MPFW. Because no standard object file format for Macintosh applications had emerged, the options for sub-application development were limited. Several possibilities were considered:

- writing MPFW as a portable C library and providing source code so that users could integrate it into the C development system of their choice. It became apparent that this was impracticable given that each development system had defined incompatible system interface files and used different stack formats, global access methods, and representations for standard types.
- requiring the use of the same proprietary development system for MPFW and sub-applications. This was regarded as an unfortunate restriction on the use of MPFW, especially since there were at the time at least six Macintosh C development systems and no clear indication of those likely to survive the competition.
- porting a public domain C compiler for use with MPFW. This was not considered viable in the context of a thesis project.
- designing a small language for implementing sub-applications. While it was acknowledged that this would require considerable effort with little original research content, the advantages of the this option were that MPFW could be presented as a self-contained package and that the linking interface to MPFW could be designed so that each sub-application could be compiled and linked independently of the framework system[1].

In the early stages of the project, the sub-application structure was intended to be similar to that of a device driver, with entry points for each event type. The limitations of this design have been discussed elsewhere in the thesis, and it was

[1]The sub-application pre-linker for Megamax object files provided this capability in the final version of MPFW.

eventually discarded in favour of multitasking, but at the time the latter approach had not been considered. The last option also gave some opportunity to see whether building the driver-like structure into a purpose-specific language would help solve some of the programming problems encountered in standard languages such as Pascal, which are primarily oriented towards hierarchical, single entry-point programs.

A prototype version of the language (IPL) was designed, and a compilation/linking system was built. A few small test programs were compiled and run on a skeletal implementation of the original MPFW. These worked satisfactorily, but at this point the language project had to be re-evaluated. Several problems were emerging:

- the compiler had already taken several months to develop, and further work was likely to endanger the other parts of the project
- the prototype language lacked structured data types, which would have been necessary for serious sub-application development. The language was potentially capable of supporting structured types, but the necessary extensions to the compiler would have required a considerable amount of work
- it was never intended that large programs should be written in IPL - single source compilation would have made this difficult; the assumption was that the MPFW shared library would provide sophisticated "building blocks" which would perform most of the complex functions required in sub-applications (screen management, text editing, file transfer, pattern matching & replacement etc.). It soon became apparent that although the MPFW library might be useful for eliminating some of the low level toolbox programming, reliance on a set of very high level packages would severely restrict the range of practical sub-application interfaces. Not only that, but development of a comprehensive high-level library might not be possible in the time scale of the project.
- the possibility of multitasking was under investigation. This would eliminate the necessity of the driver-like structure, and improve the case for using a standard programming language.

In view of these factors, it was decided to postpone further development of the compiler until MPFW and the communications protocol were fully implemented. Owing to a lack of time, the language project was eventually abandoned. No

serious sub-applications were written in IPL, nor were all features of the compiler rigorously tested. Later versions of MPFW do not support the IPL interface, and require that all sub-applications be written in Megamax C.

Nevertheless, the compiler development was an important stage in the thesis project, and a brief description of the work is therefore warranted.

IV.2 Structure of an IPL Program

An IPL program contains three divisions:

1) global data declarations
2) event handling routines
3) support routines

IPL routines are expressed in a functional prefix notation. All built-in arithmetic, logical and control functions take the same form and must be fully parenthesised. The language syntax was chosen to minimise the possibility of inadvertent ambiguities and to simplify the specifications for the parser, which was generated using YACC [JOHNSON 75] and therefore required the language to be LR(1).

IPL supports the base types character (one byte), integer (two bytes), long (four bytes) and pointer. As in C, logical operators return integer values. Arrays are supported through built-in functions on pointer types. Structured types were never implemented. Type compatibility is enforced in the parameters and return values of functions, including built-in operators, although automatic widening of operands is allowed in arithmetic expressions according to the same rules which apply to C coercions. Mismatch of parameter types has been found to be a frequent cause of error in Macintosh C programs: many toolbox and OS routines take five or more parameters, and the type checking provided by would help in the detection of such errors.

Scope and storage class rules are similar to those of C, with some restrictions. Static variables must be declared at the start of the program. They have global scope, as do all functions. Any variables declared within a function are automatic and visible only to that function. Parameters are passed by value; reference parameters must be implemented explicitly using pointers.

Routines in the event handling division are called from MPFW in response to the pre-defined events described in Chapter 5. Names and parameter types for these routines are fixed; the compiler checks them against an internal symbol table to ensure that they are called correctly from MPFW. Calls to library functions are similarly checked.

There is no provision for separate compilation of IPL programs. The compiler produces executable code resources and installs these in the sub-application file together with all support resources required for MPFW. An intermediate object file format would have required extra development which seemed unwise in such a prototype system.

IV.3 Compiler Implementation

IPL was initially envisaged as interpreted language. It was feared, however, that the performance of an interpreter acting on source or intermediate code might not be adequate for interactive applications, especially when combined with inevitable delays from the extra software layers of MPFW. Also, designing an intermediate code compiler and interpreter might well have required more effort than a native code compiler.

The implementation of IPL was the first project to make heavy use of the integrated programming environment discussed in Chapter 3. Many of the facilities described were in fact developed in direct response to problems encountered at this time. As with the other Macintosh resident programs in this thesis, the compiler was written in MEGAMAX C. The UNIX lexical analyser and parser generation tools, LEX [LESK 79] and YACC, proved invaluable in accelerating the implementation of the compiler, although the generated source code required some post-processing for compatibility with MEGAMAX C. The integrated programming environment, however, allowed full automatic project management from source generation to linkage under the control of a single MAKE script.

The generated sections of the compiler build complete memory-resident parse trees linked to the symbol table entries for each function. While a true one-pass compiler would allow larger IPL programs on systems with less memory, parameter checking in forward references to functions would require stub

declarations which I find particularly irksome. All type checking is therefore performed during tree decomposition, the type of a node being determined -

1) by the declared type of the node (for a normal function call or variable reference).
2) by the type required by the function or operator for which the current node is a parameter or operand (type information passed down the tree).
3) by the type of the parameters or operands of the current node (type information passed up the tree). This applies to arithmetic operators allowing coercion and to control flow functions such as if-then-else.

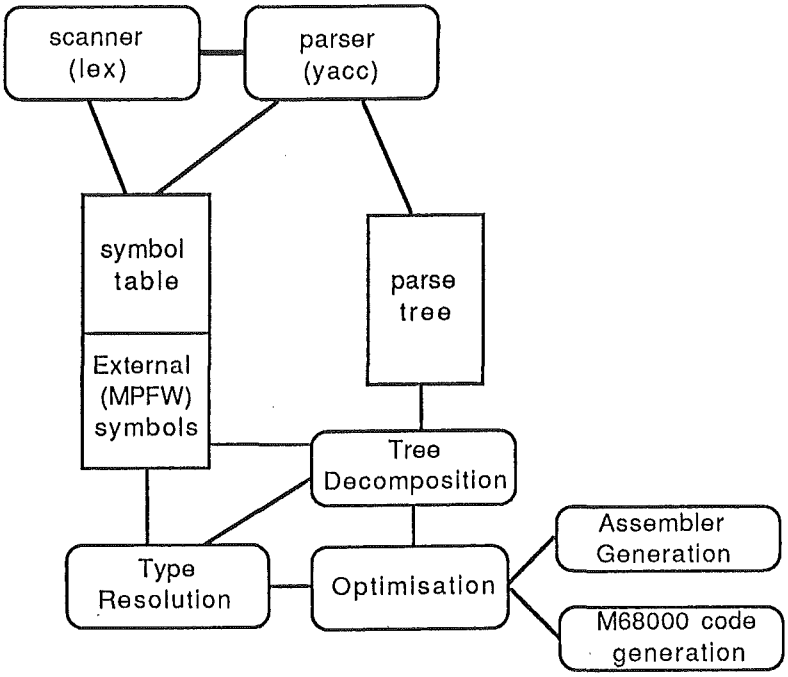
It is an error if the types imposed by these conflict.

The IPL compiler performs a small amount of code optimisation. Constant arithmetic and logical operations are reduced, as are operations involving identities. Register allocation is fairly primitive, with most registers set aside for specific purposes, although some effort is made to avoid unnecessary stacking of the operands for built-in functions.

The symbol tables for MPFW library functions and standard event handler formats are stored in the resource fork of the compiler itself and pre-loaded before a normal compilation. The compiler provides an option for creating these symbol table resources from a file of stub declarations and a link map from MPFW. This system saves the programmer from having to include system interface files with every compilation, and also allows the compiler to resolve all external references without the need for a separate linker. Pre-compiled declarations are also used in Modula-2 programming environments [GEISSMANN 83], although in these the system is designed to allow much more general integration of separately compiled units.

Because no public domain assembler for the Macintosh was available, the IPL compiler generates Motorola 68000 code directly. For debugging purposes, however, a parallel back end producing assembly language was also developed, and this proved useful in revealing subtle errors in the machine code generation. It is nevertheless a tribute to the orthogonality of the 68000 instruction set that direct code generation (through bit masking) was little more difficult than the assembly language version.

Fig 7.1 IPL Compiler Structure



Appendix V

MPFW Event Handling

Fig V.1 Main Loop

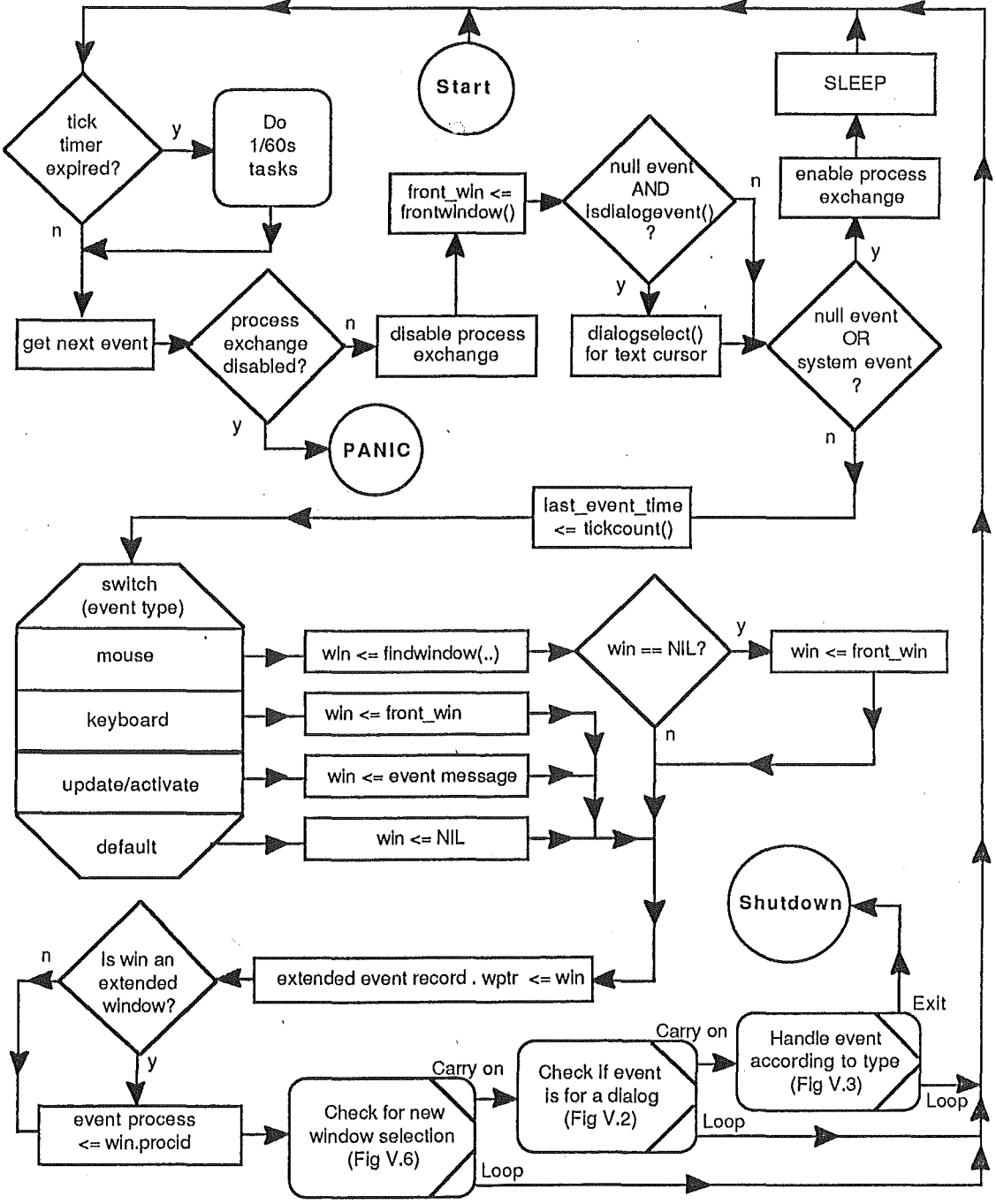


Fig V.2 Special Handling for Dialog Events

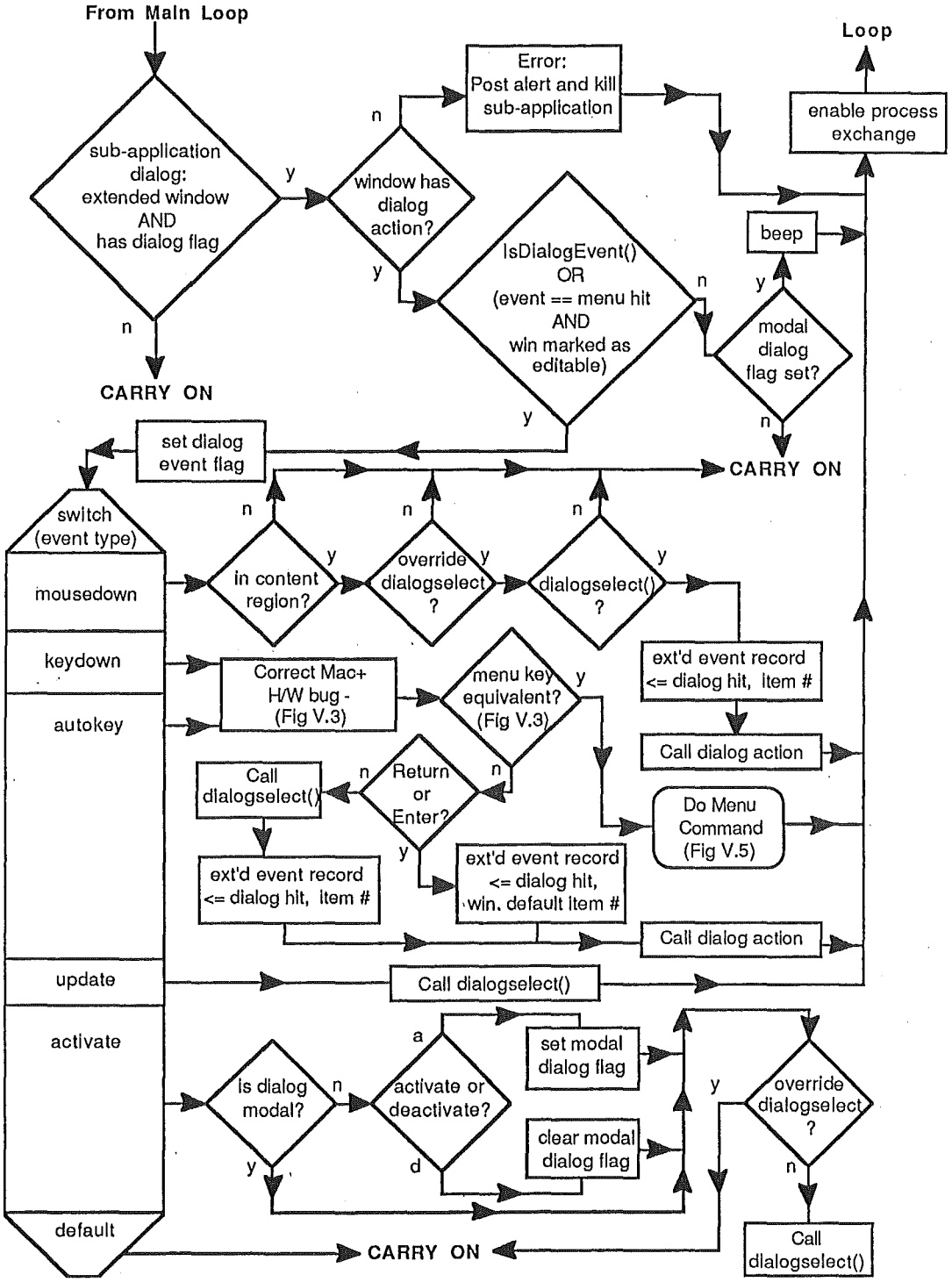


Fig V.3 Event Responses

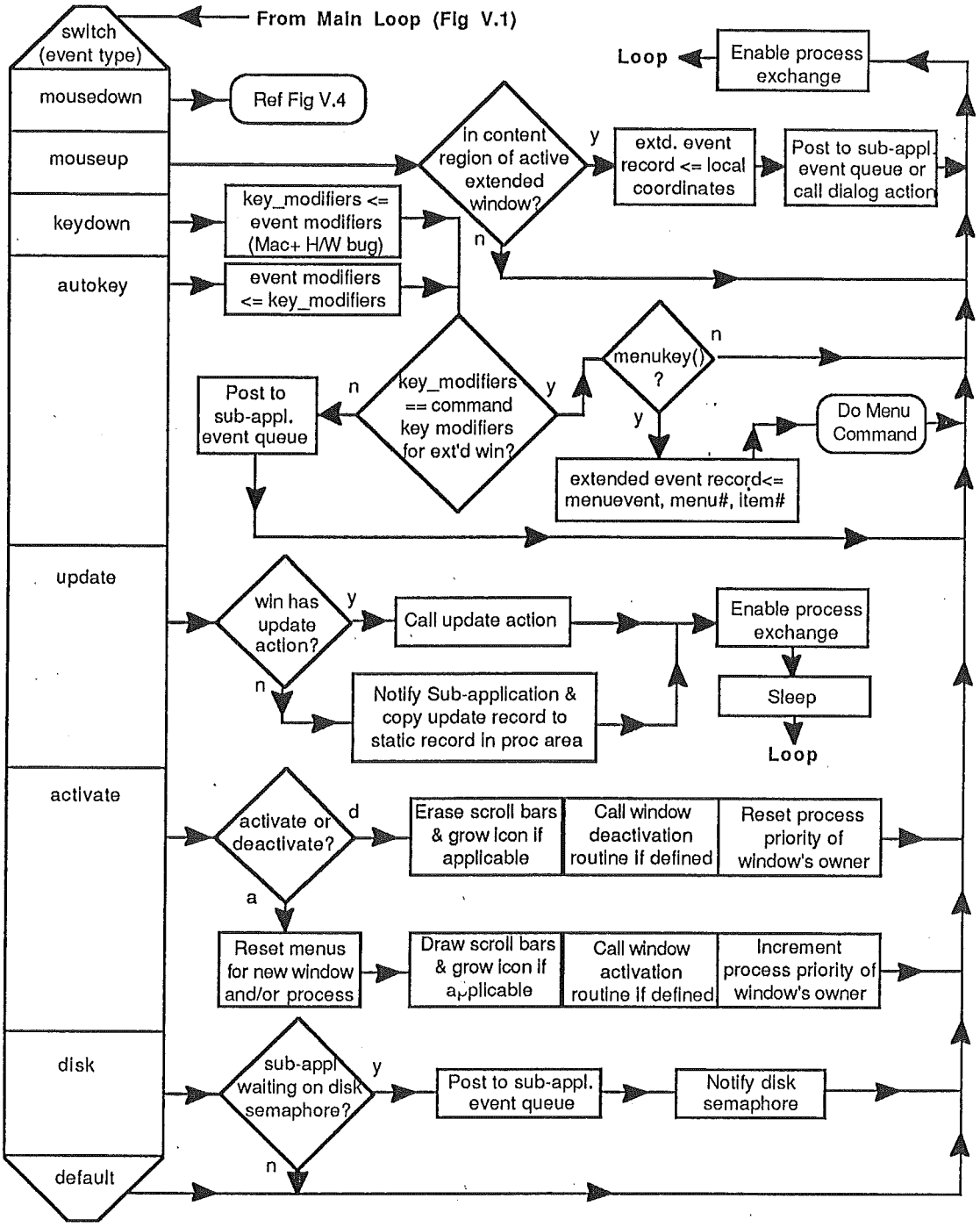


Fig V.4 Mousedown Events

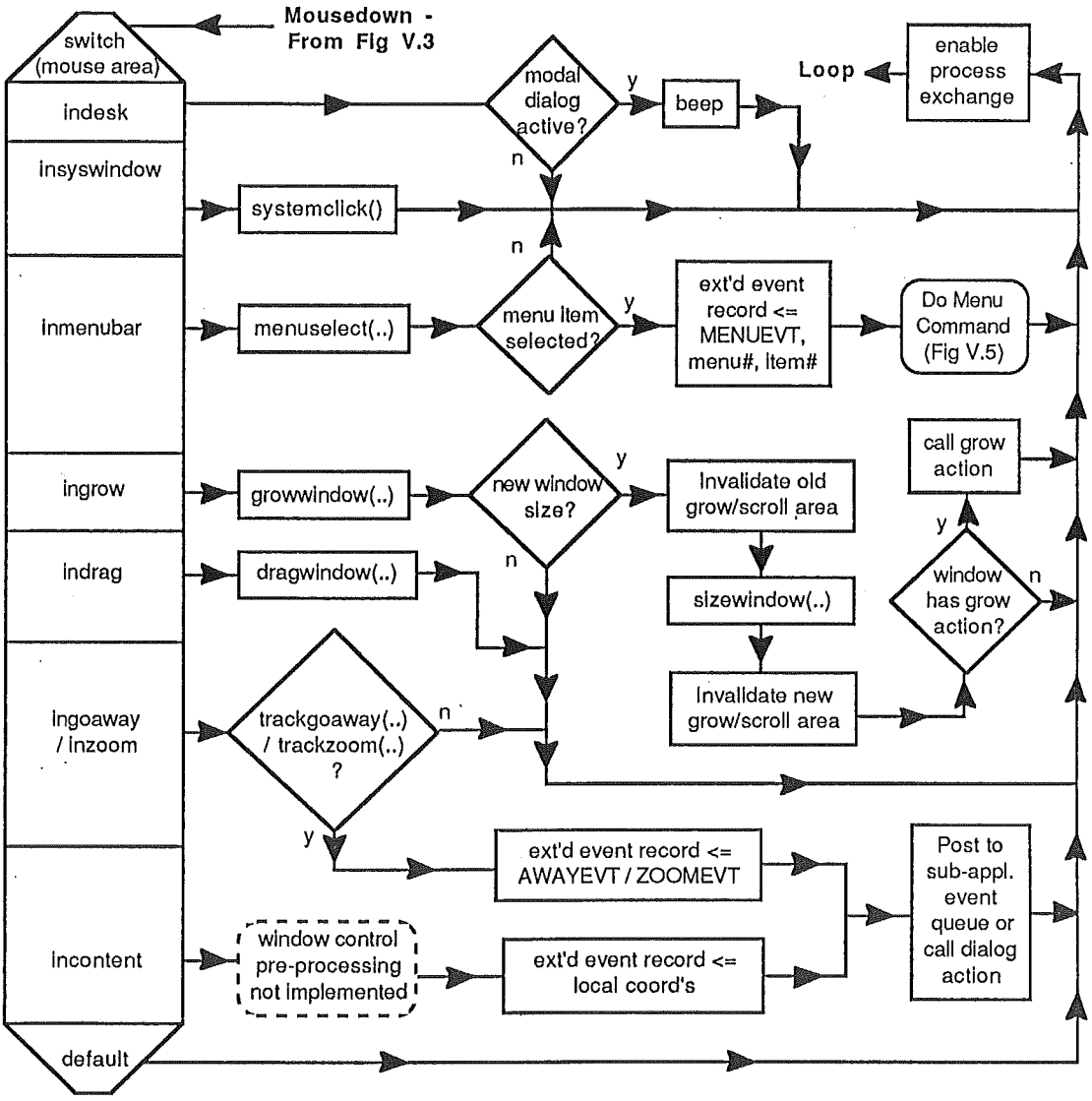


Fig V.5 Menu Actions

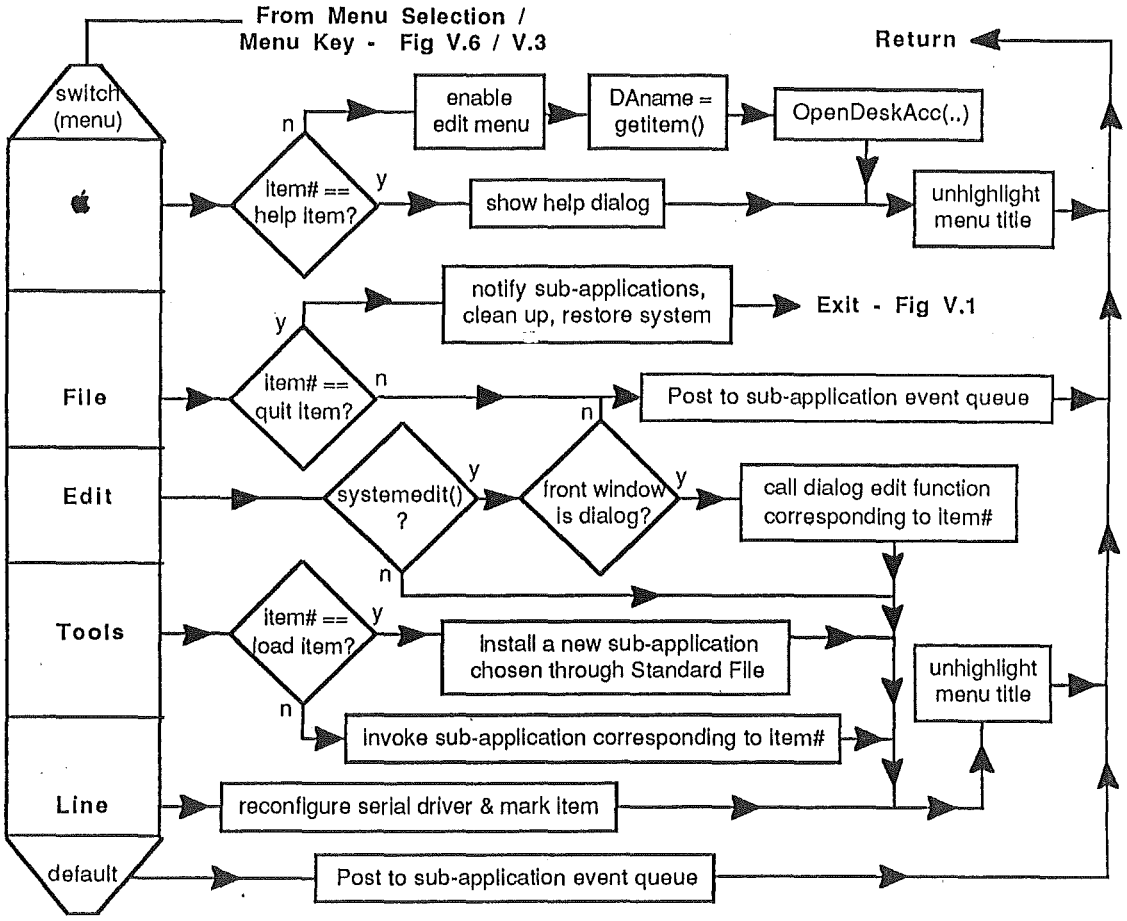
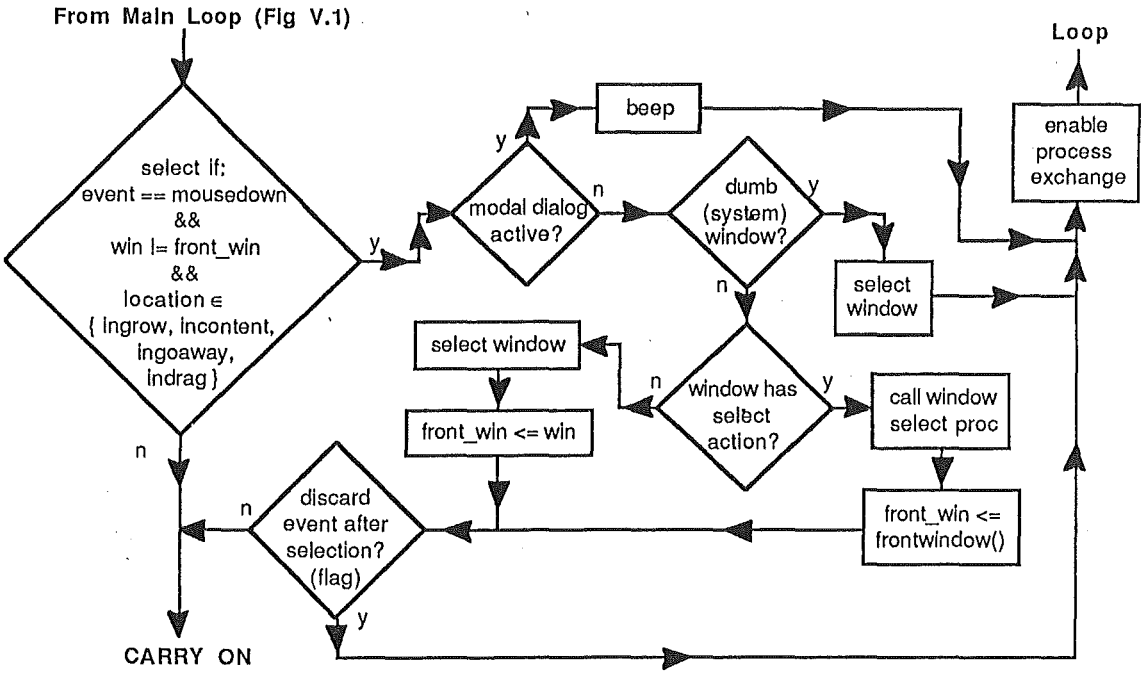


Fig V.6 Window Selection Algorithm



References

Note: Some of the following references are taken from the documentation supplied with 4.3 BSD UNIX, and share the publication information listed under the collective reference for the set below [UCB 86]. For these cases, unless an independent reference for a paper is available, the reference date applies to the appropriate manual: 1986 for BSD-specific items, and 1979 for those which date back to 7th edition or earlier.

UCB 86 [Various authors] Manuals for UNIX - 4.3 Berkeley Software Distribution, Virtual VAX-11 Version, comprising:
Unix User's Reference Manual,
Unix Programmer's Reference Manual,
Unix User's Supplementary Documents,
Unix Programmer's Supplementary Documents, vols.1&2,
User Contributed Software,
Unix System Manager's Manual,
Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley.
Regents of the University of California, 1979/80/83/86.
A.T.&T. Bell Laboratories, Inc., 1979.

AHO 79 Aho, A.V., Kernighan, B.W, Weinberger, P.J. Awk - A Pattern Scanning and Processing Language (Second Edition). *UNIX User's Supplementary Documents*.

APPLE 85 Apple Computer, Inc. *Inside Macintosh*, vols. I-III. Reading, Mass., Addison-Wesley, 1985.

APPLE 86 Apple Computer, Inc. *Inside Macintosh*, vol. IV. Reading, Mass., Addison-Wesley, 1986.

APPLE TN Apple Programmers' and Developers' Association. Macintosh Technical Notes 1986-1988.

BELL 78 Ritchie, D.M., Thompson, K., and others. (Various papers). *Bell Systems Technical Journal* vol. 57, no. 6; June 1978.

BRUNER 85 Bruner, J.D. UW : A Multiple-Window Terminal Emulator for use with 4.2BSD UNIX. Lawrence Livermore National Laboratory, Livermore, C.A., 1986. Paper supplied with software distribition.

BUXTON 83 Buxton, W. and others. Towards a Comprehensive User Interface Management System. *Computer Graphics* vol.17, no.3, pp35-43; July 1983.

CARGILL 83 Cargill, T.A. The Blit Debugger. *The Journal of Systems and Software* vol.3 pp277-284; 1983.

FELDMAN 79 Feldman, S.I. Make - A Program for Maintaining Computer Programs. *UNIX Programmer's Supplementary Documents*, vol.1.

GEISSMANN 83 Geissmann, L.B. *Separate Compilation in Modula-2 and the Structure of the Modula-2 Compiler on the Personal Computer Lilith.* Zurich, Dieser Publications, 1983.

GETTYS 85a Gettys, J. Problems Implementing Window Systems in UNIX. MIT Project Athena, Massachusetts, 1986. (USENIX paper supplied with **X** software distribution)

GETTYS 85b Gettys, J., Newman, R. & Della Fera, T. Xlib - C language X Interface Protocol Version 10. MIT Project Athena, Massachusetts, 1986. (Supplied with **X** distribution)

GOSLING 85 Gosling, J.D. SunDew - A Distributed and Extensible Window System. In *Methodology of Window Management.* pp47-58. Hopgood, F.R.A. et. al., eds., New York, Springer-Verlag, 1986.

GREENBERG 85 Greenberg, S. and Witten, I.H. Interactive End-User Creation of Workbench Hierarchies Within a Window System. *CIPS Session Conference Papers,* Toronto, 1985.

HANSEN 84 Hansen, S.J. and others. Interface Design and Multivariate Analysis of UNIX. *ACM Transactions on Office Automation Systems,* vol.2, no.1, pp42-58; January 1984

HILL 84 Hill, D.R., Witten, I.H., Neal, R., Lomow, G., JECL and HIDE: Practical Questions for the JADE User Interface *CIPS Session Conference Papers,* Calgary, 1984.

HILL 86 Hill, D.R. Interacting with Future Computers *ICFAC Conference Proceedings ,* Christchurch, February 1986.

ISO 82 International Standards Organisation. High-level Data Link Control Procedures. ISO 3309 (Part 1) and ISO 4335 (Part 2), 1982.

JACOB 80 Jacob, R.J.K. User Level Window Managers for UNIX. *Proceedings of the UniForun International Conference on UNIX,* 1980.

JACOBS 85 Jacobs, C.J.H. YAP -Yet Another Pager. Vrije Universitiet, Amsterdam. (Software distribution, 1985)

JOHNSON 75 Johnson, S.C. Yacc - Yet Another Compiler Compiler. (Computer Science Technical Report 32, Bell Laboratories, 1975) *UNIX Programmer's Supplementary Documents,* vol.1.

JOHNSON 79 Johnson, S.C. Lint - A C Program Checker. *UNIX Programmer's Supplementary Documents,* vol.1.

JOY 86a Joy, W.N. An Introduction to the C-Shell. (revised Seiden, M. 1986). *UNIX User's Supplementary Documents.*

JOY 86b Joy, W.N. and Horton, M. An Introduction to Display Editing with Vi. *UNIX User's Supplementary Documents.*

KERNIGHAN 79 Kernighan, B.W. A Tutorial Introduction to the UNIX Text Editor. *UNIX User's Supplementary Documents.*

KNASTER 86 Knaster, S., *How to Write Macintosh Software.* Hasbrouck Heights N.J., Hayden Book Co./Apple Press, 1986.

LANTZ 86a Lantz, K.A., Multi-process Structuring of User-Interface Software. *Computer Graphics,* vol.21 no.2 pp124-130; April 1987.

LANTZ 86b Lantz, K.A. and others. Reference Models, Window Systems and Concurrency. *Computer Graphics,* vol.21 no.2 pp87-97; April 1987.

LESK 79 Lesk, M.E., and Schmidt, E. Lex - A Lexical Analyser Generator. *UNIX Programmer's Supplementary Documents,* vol.1.

MCMAHON 79 McMahon, L.E. SED - A Non-interactive Text Editor. *UNIX User's Supplementary Documents.*

MORRIS 86 Morris, J.H. and others. Andrew: A Distributed Personal Computing Environment. *Communications of the ACM,* vol.29, no.3; March 1986.

PERLMAN 81 Perlman, G. Menunix: A Menu-Driven Interface to UNIX Programs. Cognitive Science Laboratory, University of California, San Diego, 1981. (Paper supplied with MENUNIX software distribution)

PIKE 84 Pike, R. The Blit: A Multiplexed Graphics Terminal *A.T.&T. Bell Laboratories Technical Journal* vol. 63 no. 8 part II; October 1984

SCHEIBELHUT 79 Sheibelhut, D.M. A Visual Command Processor for the UNIX Operating System. University of California, Berkeley, 1979. (Paper supplied with VSH software distribution).

SHOENS 86 Shoens, K. Mail Reference Manual. (revised Leres, C. 1986) *UNIX User's Supplementary Documents.*

SUN 86a Sun Microsystems, Inc. *Programmer's Reference Manual for the Sun Windows System.* SMI, 2550 Garcia Ave, Mountain View, CA. 1986.

SUN 86b Sun Microsystems, Inc. Network File System Protocol Specification. In *Networking on the Sun Workstation.* SMI, 2550 Garcia Ave, Mountain View, CA. 1986.

SUN 86b Sun Microsystems, Inc. RPC Protocol Specification, §3.2. In *Networking on the Sun Workstation.* SMI, 2550 Garcia Ave, Mountain View, CA. 1986.

SUN 87 Sun Microsystems, Inc. *NeWS Manual.* SMI, 2550 Garcia Ave, Mountain View, CA. 1986.

TANNER 86 Tanner, P.P. and others. A Multitasking Switchboard Approach to User Interface Management. *Computer Graphics,* vol.20 no.4; August 1986.

THIMBLEBY 85 Thimbleby, H. Failure in the Technical User Interface Design Process. *Computers and Graphics* vol. 9, no.3, pp35-43; 1985.

TRUSCOTT 85 Truscott, T. and Lennon, M. WM. Research Triangle Institute, North Carolina, 1985. (Software distribution)

TUCKER 86 Tucker, A.B. *Programming Languages.* pp59-60 & 507-8. 2nd. Ed. New York, McGraw-Hill, 1986.

UCB 86a Joy, W.N., Fabry, R.S., McKusick, M.K., Karels, M. Berkeley Software Architecture Manual - 4.3BSD Edition *UNIX Programmer's Supplementary Documents,* vol.1.

UCB 86b Leffler, S.J., Fabry, R.S., Joy, W.N., Lapsey, P., An Advanced 4.3 BSD Interprocess Communication Tutorial *UNIX Programmer's Supplementary Documents,* vol.1.

UCB 86c *UNIX Programmer's Reference Manual* (Sections 2, 3, 4 & 5)

WEISEN 83 Weisen, M. and others. The Maryland Window Systems. Computer Science Dept., University of Maryland, 1983. (Paper supplied with software distribution)

WIRTH 82 Wirth N. *Programming in Modula-2.* 2nd Ed. New York, Springer-Verlag, 1982.